A STUDY INTO CONCURRENT ADVANCEMENT OF SIMULATION AND EMULATION USING COMPOSITE SYNCHRONIZATION.

BY

VLADIMIR ADAM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor David M. Nicol

ABSTRACT

Simulating a communication network is often a necessity before deploying it into the real world. However, behavior of purely simulated models is limited by the functionality of the network simulator. Instead, emulation can be combined with simulation to simulate real traffic generated by real applications. This integration of simulation and emulation is done in virtual time thanks to TimeKeeper, a recent open source tool that brings Linux processes to virtual time. Previous work involved integration of TimeKeeper with simulators such as CORE and ns-3.

The purpose of this thesis is to explore and analyze the integration of TimeKeeper and S3FNet and closely couple simulation and emulation. Previously, S3FNet was combined with emulation via OpenVZ containers and synchronized simulation and emulation using stop-and-go barrier based synchronization. This work combines S3FNet with Linux containers and utilizes composite synchronization to achieve a tighter coupling between simulation and emulation. This thesis explores the challenges and limitations of emulation with Linux containers after their integration with S3FNet.

To my parents and grandparents, for their neverending love and support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
DML	Domain Modeling Language
LXC	Linux Container
PID	Process Identifier
RTT	Round Trip Time
SSF	Scalable Simulation Framework
S3F	Simpler Scalable Simulation Framework
TDF	Time Dilation Factor

CHAPTER 1

INTRODUCTION

1.1 Motivation

In the effort to study and analyze the behavior of communication networks, real world networks are modeled and simulated to reduce costs and to increase fidelity. Large models of networks can be simulated in a small and controlled environment which empowers the modeler with control and understanding of a network's behavior. These simulations can either be completely deterministic and repeatable, or *almost* deterministic and repeatable. Almost signifies that the variance in the repeatability of such simulations can be small. Simple Scalable Simulation Framework (S3F) [14] is a recent update of the Scalable Simulation Framework (SSF) [7] API that allows for simpler creation of deterministic simulation models of real world networks. S3FNet, a parallel discrete-event network simulator created using the S3F API, allows for creation of deterministic simulations of network models which often consist of hosts and routers. S3FNet can also simulate non-deterministic emulated hosts which generate traffic created by real applications. Both approaches are attractive even though emulation requires more resources per unit model than simulation. The overall goal of this thesis is the exploration and analysis of integrating lightweight simulation with lightweight emulation.

1.2 Background

1.2.1 Linux Containers (LXCs)

An LXC [3, 9] is a lightweight Linux system container that behaves closely to an actual virtual machine which in turn can behave like a physical host. LXCs utilize Linux kernel security features like namespaces, control groups, and mandatory access controls to group and contain resources into containers. Specifically, a container isolates a part of a host's resources in order to run an application or system inside it.

There are three types of virtualizations: emulation, paravirtualization, and operating system-level virtualization [6]. Emulation of hardware allows multiple and different operating systems to run as long as the host operating system supports the platform which it emulates. Emulation has the lowest density and lowest performance as it is the most resource demanding. Paravirtualization allows a host to run modified operating systems on top of a controller called hypervisor which monitors and manages virtual machines. Paravirtualization has higher density and performance than emulation. Lastly, operating system-level virtualization uses the host operating system to create containers and has the highest density and performance.

LXCs are realized through operating system-level virtualization via a virtual environment. This allows many LXCs to run on a single host machine where each LXC shares the host's operating system kernel. However, a potential downside with this type of virtualization is that the host virtual machines can only virtualize one type kernel as opposed to multiple types.

An LXC is essentially is a group of processes under a single network namespace giving the illusion of a virtual machine. Specifically, an LXC consists of a parent process that spawns children processes each of which can run

multiple applications in an isolated and contained environment. LXCs are actively being developed, maintained, and pushed into the mainstream Linux kernel. This, along with the fact that they provide lightweight virtualization, makes LXCs an attractive option for combining simulation and emulation. Like previous work which integrated emulated hosts with virtual time, LXCs also need to maintain a notion of virtual time. This is accomplished thanks to TimeKeeper, a tool that brings virtual time to Linux processes.

1.2.2 TimeKeeper

TimeKeeper [13, 12] is a Linux kernel module that allows Linux processes to maintain a notion of virtual time. Since an LXC is a collection of processes, TimeKeeper can be particularly integrated with LXCs. TimeKeeper achieves this by slightly modifying the Linux 3.10.9 kernel to allow *dilated* processes to operate in virtual time. TimeKeeper gives a dilated process a notion of virtual time by modifying the Linux *gettimeofday(...)* function and associating a time dilation factor (TDF) with a Linux process.

A process with a TDF τ, such that $\tau \neq 1$, is dilated and *gettimeofday(...)* returns to the calling process its virtual time as a function of τ and wallclock time elapsed since the process was dilated. Specifically, consider a process dilated at wallclock time W_1 with TDF τ. A *gettimeofday(...)* call at W_2 returns $W_1 + ((W_2 - W_1) \times \tau)$. An LXC is dilated by dilating an LXC's parent process *as well* as all of its children processes. In order to accurately advance a process in virtual time, TimeKeeper controls the scheduling of a dilated process by allowing that process to run for a specific amount of wallclock time without preemption. This increases the preciseness of allocating CPU time to a process which in turn increases the accuracy of the modified

gettimeofday(...).

Section 1.2.2.1 explains the concept of virtual time in more detail. TimeKeeper was designed as a lightweight and simple solution to allow for easy integration of emulation with already existing network simulators. TimeKeeper's API contains three key functionalities:

- *freeze(PID)*: freezes a process immediately and prevents further advancement and execution. Since an LXC is a collection of processes, an LXC is frozen by freezing the LXC parent process as well as all of its children processes.

- *unfreeze(PID)*: unfreezes a process and immediately resumes advancement and execution. Like with *freeze(...)*, an LXC and its children processes would be unfrozen at this time.

- *dilate(PID, TDF)* dynamically changes the TDF of a process. Note - all processes start undilated with a TDF of 1.

Combining these functionalities together allows TimeKeeper to control, with great accuracy, the advancement of LXCs in virtual time. However, TimeKeeper cannot perfectly control an LXC's advancement due to the granularity of Linux's timer. Furthermore, the amount on machine instructions executed when an LXC attempts to emulate identical behavior may vary. Specifically, an LXC is scheduled to run for T units of wallclock time but instead runs for $T \pm \epsilon$ due to the nature of the operating system scheduler and overhead.

1.2.2.1 Virtual Time

Virtual time allows for the scaling of the performance of a process. In other words, virtual time can seemingly make an application advance faster or

slower. For instance, a process performs 100 seconds of computation in wallclock time but represents that time by 50 virtual seconds. To an outsider looking at a process's virtual time, the process performs 100 real seconds of computation in half the time. The same strategy can be applied in reverse, making a process seem slower. Now, a process performs 100 seconds of computation in wallclock time and advances by 200 virtual seconds. TimeKeeper achieves this by assigning a TDF τ to each Linux process which changes the advancement of that process's virtual time by τ. Specifically, $\tau > 1$ gives the illusion that a process performs faster while $\tau < 1$ gives the illusion that a process performs slower. For example, a process with $\tau = 1.5$ advances its virtual time by 2 virtual seconds for every 3 seconds of wallclock time. Similarly, a process with $\tau = 0.5$ halves the advancement of virtual time by advancing virtual time by 1 virtual second for every 0.5 seconds of wallclock time. Intuitively, when the $\tau = 1$, a process is not dilated and virtual time advances at the same rate as wallclock time.

1.3 Simulators and Emulators

In many research areas, simulation of network models is lucrative since both small and large experiments can be repeated. One of TimeKeeper's main objectives was to use something else in place of OpenVZ containers. LXCs fit well into this objective and TimeKeeper proved its generality by integrating itself with two already existing and very different network simulators: ns-3 [5] and the Common Open Research Emulator (CORE) [1]. Both integrations required minimal changes to CORE and ns-3 which is unlike S3FNet's integration with TimeKeeper.

1.3.1 CORE

CORE [8, 12] is an open source hybrid simulator which only emulates the networking stack of hosts with lightweight virtualization and simulates the wires between these hosts. Specifically, CORE focuses on emulation of network layer 3 and above. Similar to an LXC, CORE uses network namespaces but does not explicitly use LXCs. In [12], TimeKeeper was integrated with CORE in order to increase the accuracy of CORE's behavior.

1.3.2 Ns-3

Ns-3, a hybrid discrete event simulator designed for educational and research use, can model both simulated and emulated hosts. Ns-3 can interact with real systems, however, it is limited by the size of the network model. Like CORE, ns-3 was also integrated with TimeKeeper [12] to allow for simulation and emulation of larger models. To reiterate, the main topic of this thesis is the exploration and integration of another discrete event simulator, S3FNet, with TimeKeeper.

1.4 S3F

S3F is a second generation API based on the Scalable Simulation Framework (SSF). SSF was originally intended to improve performance of discrete-event simulations by parallelizing simulations of complex network models. In [14], Nicol, Jin, and Zheng revisit SSF and redesign it to create easier ways of exploiting potential parallelism. S3F relies primarily on standard C++ libraries which attests to its simplicity. S3F shares with SSF the notion entities, inChannels, outChannels, processes, and events. S3F adds on the concept of

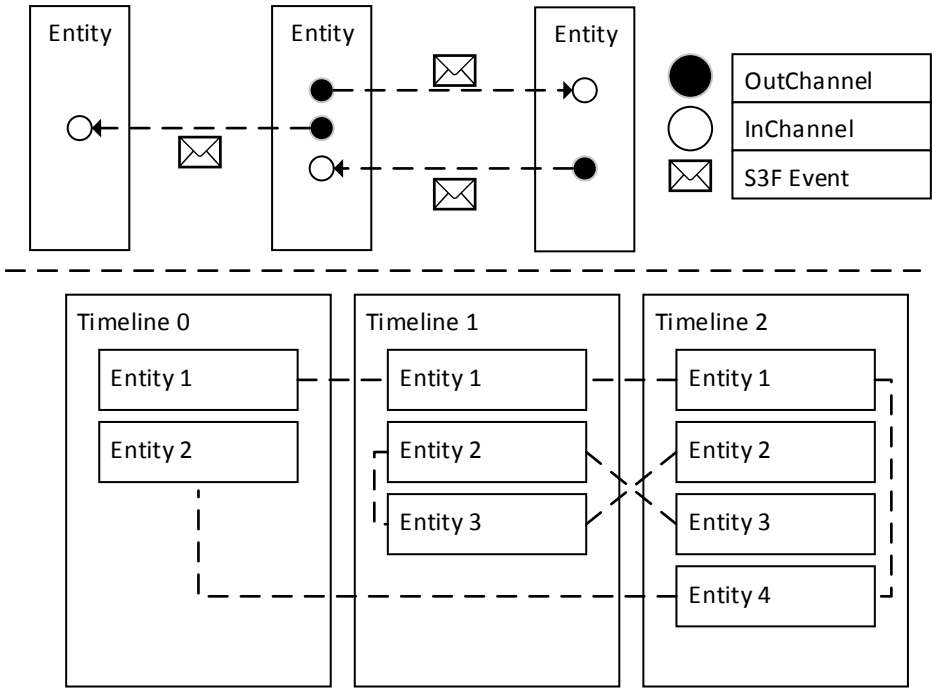

Figure 1.1: S3F Architecture

"message" and "interface" and makes the timeline accessible to the modeler. The big picture is that entities, aligned on timelines, send events across channels. This is showcased in Figure 1.1.

A crucial and key component of the S3F API is the timeline class which maintains virtual time and manages event lists. A timeline can manage any number of entities as long as each entity belongs to exactly one timeline. Each timeline also maintains and updates its own event list which is updated with generated events as the simulation advances forward in virtual time. Specifically, a timeline advances its own virtual time through temporal changes in events. If a timeline is at virtual time T_a and the next event to simulate has timestamp $T_b > T_a$, then the timeline advances its virtual time to T_b and processes the event. Each timeline is represented by a *pthread* and controlled by the *Interface* class. The *Interface* class manages timelines and

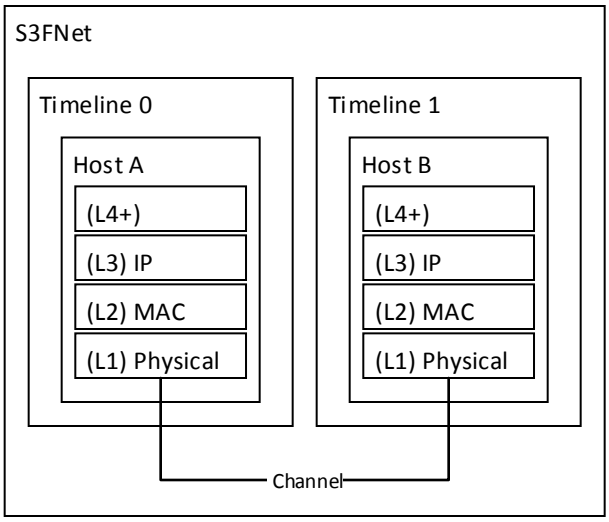

Figure 1.2: Simplest S3FNet Simulation

facilitates the advancement and halting of simulations.

1.5 S3FNet Overview

S3FNet [10] is a parallel network simulator based on the S3F kernel much like SSFNet is based on the SSF kernel. S3FNet can simulate large scale network models consisting of entities which represent hosts or routers. Network models are represented using the Domain Modeling Language (DML) configuration files. The most basic simulation, showed in Figure 1.2, consists of a single network which contains 2 hosts. In it, 2 hosts aligned on 2 unique timelines are simply connected with one another. S3FNet supports 2 types of synchronization techniques of timelines during simulation advancement. These will be discussed in Section 2.4. Currently, there exist two S3FNet versions: a base version that models purely simulated networks and a full version that adds emulation capabilities through OpenVZ containers.

1.5.1 Base S3FNet

The main feature of base S3FNet is parallel discrete-event simulation. Using S3FNet, a modeler can create large network models consisting of hosts and routers to discretely simulate a network scenario. S3FNet includes the most common network topology concepts such as TCP/UDP clients/servers. Other constructs and protocols such as ICMP, however, have to be added through an easy-to-extend S3FNet API. An S3FNet simulation must have at least one timeline. Consequently, an S3FNet simulation will always consist of at least two threads: a single control thread responsible for advancing timelines through epochs and one thread for each timeline which allows to exploit parallelism. The control thread instructs timelines to enter the simulation state and once those timelines finish an epoch, the control thread regains control. At this point, the control thread can modify the simulation model, for example, by removing a router or adjusting a link's speed. The base S3FNet version is limited by the fact that traffic inside a base S3FNet simulation is completely simulated. However, real traffic can be emulated using real applications running inside virtual machines through a solution called S3FNet-OpenVZ [16, 11, 10].

1.5.2 S3FNet-OpenVZ

S3FNet with OpenVZ contains all the features of the base S3FNet version but also adds emulation support by using OpenVZ [6] containers. This allows for simulation of real traffic generated inside OpenVZ containers. To embed OpenVZ containers with virtual time, the OpenVZ kernel and scheduler were modified to add the ability to run OpenVZ containers for a burst of virtual time and stop. However, as described in [16], the actual amount of

execution time can vary from the desired amount of execution time which introduces the challenge of deterministically repeating a timeslice of emulation time. This problem, however, does not exist in base S3FNet as the purely simulated simulations are repeatable and fully deterministic. It is important to note that S3FNet-OpenVZ is limited to very specific versions of the Linux kernel as OpenVZ containers only operate on certain Linux kernels. This restricts the flexibility of systems used for running hybrid simulation experiments. Integration of S3FNet with LXCs will not have such a limitation due to the state of development of LXCs mentioned in Section 1.2.1. If TimeKeeper successfully integrates with the mainstream Linux kernel, the above restriction would significantly loosen.

CHAPTER 2

S3FNET WITH TIMEKEEPER

The question arises: what to call this new solution? Following the name convention set up for the previous 2 iterations of S3FNet, and due to the main appeal of LXCs, the name S3FNet-LXC seems appropriate.

2.1 Motivating Scenario

Consider a research experiment to evaluate the potential feasibility of deploying 10,000 Linux servers connected through 5,001 routers. It would be difficult and costly to replicate this using physical hardware in a research lab. Instead, it is possible to model and simulate this scenario in software. This scenario, shown in Figure 2.1, is modeled using hosts and routers inside an S3FNet simulation. Each simulation host has a corresponding LXC which can model the behavior of a single server. In this case, each LXC will generate Ethernet packets that will ultimately get sent to other LXCs via the S3FNet simulator instead of going directly to the destination LXC. The simulator can control how long it takes for a packet to travel between the LXCs. A modeler may want to know what happens in the simulation if one link between a server and a router has a 10 μs propagation delay instead of a 100 μs propagation delay. Or perhaps, the modeler is interested to see what happens when a packet is occasionally dropped. S3FNet can simulate these possible real life situations. To make this reality, S3FNet must synchronize

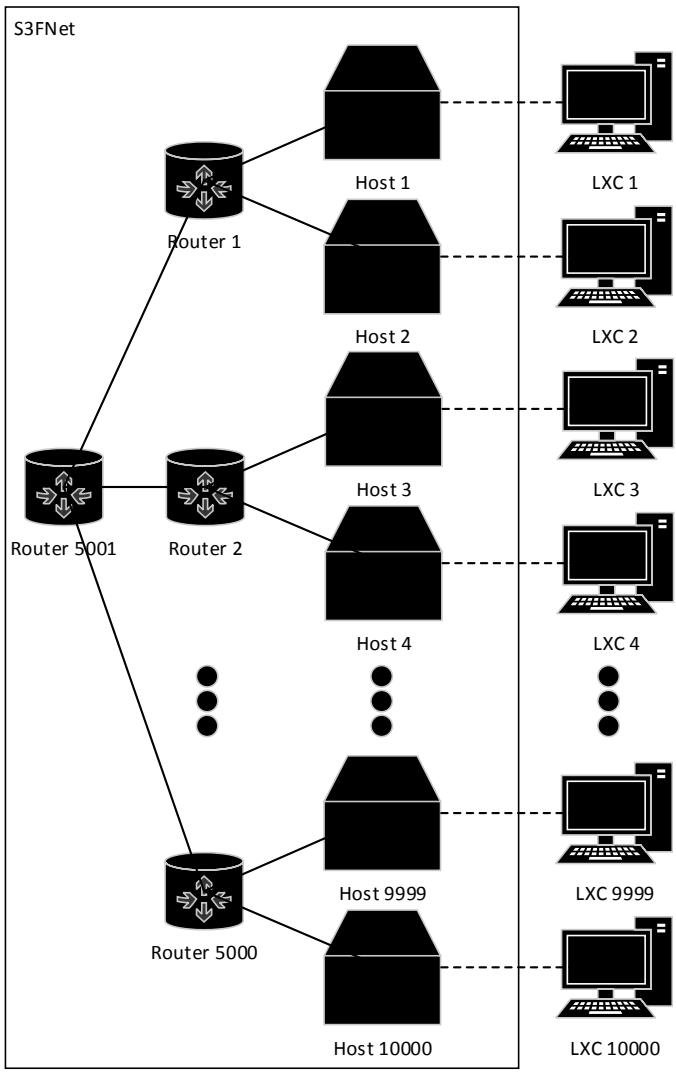

Figure 2.1: Motivating Example Modeling 100,000 LXCs

itself with the LXCs in virtual time. This synchronization and facilitation of packets between LXCs must be done carefully in order to ensure the preciseness and accuracy of the simulation. The challenges that were overcome in accomplishing this integration is another main focus of this thesis and will be discussed in subsequent sections.

2.2 Development Process

The first step was to learn all about both the base S3FNet version and S3FNet-OpenVZ version and understand changes made to base S3FNet to realize S3FNet-OpenVZ. The second step was to understand the two kinds of synchronization techniques used in S3F: barrier-based synchronization and composite synchronization [15]. Similar to the S3FNet-OpenVZ implementation, S3FNet-LXC contains a separate module called *LXC Manager* which is responsible for the facilitation of packets in and out of LXCs. A more detailed explanation of the LXC Manager will be provided in Section 2.9.

2.3 Design Goals

With several already existing solutions for combined simulation and emulation such as ns-3 [5] and S3FNet-OpenVZ [16], this thesis explores the integration of base S3FNet with TimeKeeper with 3 unique design goals.

Goal 1: The first goal is to utilize LXCs in place of OpenVZ containers, as LXCs are slightly more lightweight and portable. Furthermore, a single host can run thousands of LXCs as opposed to hundreds of OpenVZ containers. The purpose is to integrate TimeKeeper and LXCs with S3FNet to create S3FNet-LXC.

Goal 2: The second goal is to facilitate communication between LXCs and S3FNet-LXC by using TUN/TAP devices which allow for fast transfer of packets between processes, as opposed to going through the Linux network stack.

Goal 3: The last goal is to explore a tighter coupling between emulation and simulation by using *composite synchronization* [15]. This tighter cou-

pling is realized thanks to TimeKeeper's ability to individually manage and advance LXCs.

Previously with S3FNet-OpenVZ, simulation and emulation advanced in stop-and-go fashion. Specifically, S3FNet-OpenVZ had an emulation phase during which traffic was collected from real applications running on OpenVZ containers in virtual time. All these OpenVZ containers executed applications for a specific and predetermined *timeslice*. After the emulation phase ended, this traffic was injected and simulated using S3FNet. After the simulation phase ended, the emulation phase began once again. This alternation of simulation and emulation continued until the simulation ended. Additionally, S3FNet-OpenVZ only allowed emulation and simulation to communicate at barrier synchronization points. S3FNet-LXC allows for a tighter connection between simulation and emulation and gives the illusion that both emulation and simulation advance concurrently. Instead of advancing all LXCs by a single timeslice, it is possible to loosen this restriction and advance individual LXCs by different timeslices so that LXCs synchronize more closely with S3F timelines.

2.3.1 LXC

As mentioned in Section 1.2.2, TimeKeeper brings virtual time to LXCs, making them an ideal candidate for integration with S3FNet. During the development of S3FNet-LXC, LXC 0.7.5 was used, however, the most recent and current version is LXC 1.1.0. This further attests to the fact that LXCs have a future in Linux.

In addition to the typical and common IPv4 packets which travel between LXCs, LXCs often send out packets that have to be filtered out. Such

packets include the DHCP Bootstrap protocol, or the IPv6 protocol. These packets often occur during a host's boot process and are therefore ignored in the simulation as the LXCs emulate physical machines that, in the real world, would have already been running for some time. On top of that, S3FNet does not currently support IPv6 addressing.

2.3.2 TUN/TAP Devices

TUN and TAP [4] devices are virtual-network kernel devices. These devices differ from everyday network devices as they are implemented in software and do not require actual physical hardware. TUN devices work with layer 3 to capture and send IP packets. On the other hand, TAP devices work with layer 2 to capture and send Ethernet frames.

User-space programs can attach to these kernel devices which allows them to read and write Ethernet packets to and from Linux applications via file descriptors. This is useful because S3FNet-LXC can attach itself to LXCs to read and write Ethernet packets efficiently. Since the goal is to model real network traffic, TAP devices are used in place of TUN devices.

Figure 2.2 showcases how a TAP device is connected to an LXC. The virtual Ethernet interface inside an LXC attaches itself to a Linux bridge. This Linux bridge is connected to a promiscuous TAP interface which listens to any packets passing through it. When an LXC sends out a packet through its virtual interface, it gets forwarded to the Linux bridge. The promiscuous TAP-device captures this packet and injects it into a user-space program which is listening. This Linux bridge is completely separate from the operating system network and does not know how to handle an LXC's packet. The bridge simply forwards them to the TAP device. This is useful since the goal

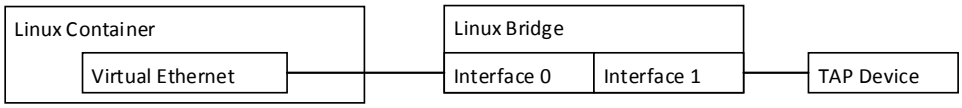

Figure 2.2: LXC Connected To Linux Bridge Via Promiscuous TAP Device

is to prevent a packet from going *directly* between LXCs. S3FNet-LXC needs to have this ability to be the middleman and oversee every packet generated by LXCs.

2.4 S3FNet Scheduling and Synchronization

S3FNet uses two types of synchronization techniques: barrier-based synchronization and composite synchronization [15]. With barrier-based synchronization, timelines compute a barrier which is the earliest time that a timeline can be affected by an event generated by a different timeline. Next, timelines in the simulation advance concurrently until they reach this barrier at which point they calculate a new barrier. Consider a concrete scenario. Suppose that there are two entities, E_1, and E_2, assigned on timelines T_1 and T_2 respectively. The total delay between them (propagation delay *and* transfer delay) is D. A message generated from E_1 at time J will not affect E_2 until time $J + D$. Assuming $J + D$ is the cross-timeline channel between T_1 and T_2, T_1 and T_2 will synchronize at $J + D$. This barrier-based synchronization was primarily used in the integration of S3FNet with OpenVZ containers.

Composite synchronization is a combination of both barrier-based synchronization and appointment-based synchronization. With composite synchronization, each cross-timeline channel between entities is labeled as either

fast or *slow*. This depends on the synchronization window of size ω, which is determined by the composite synchronization algorithm. Currently in the base S3FNet version, the synchronization window ω is heuristically defined by taking the minimum of all cross-timeline channel delays, "min_{xtd}" and multiplying it by the total number of timelines in an S3FNet simulation "num_{TL}" to get $\omega = (min_{\text{xtd}} \times num_{\text{TL}})$. If a cross-timeline channel's latency $D > \omega$, then the cross-timeline channel is *slow*. Otherwise, the cross-timeline channel is *fast*. The simulator simulates events and advances timelines in virtual time based on the latency of a channel between two entities. Consider the two possible cases for cross-timeline channels.

Cross-timeline channel is slow: Events passing across slow channels will not be delivered until timelines reach ω. A message on a slow channel will not affect the recipient until a time greater than ω which means that the simulator does not process that message across the slow channel immediately. A timeline schedules a timeout event which will arrive on the destination timeline at its expected arrival time, after all timelines reach ω.

Cross-timeline channel is fast: Events passing across fast channels must be synchronized dynamically within ω. For instance, consider an event being sent from timeline T_1 to timeline T_2 with a cross-timeline channel delay of D. T_1 and T_2 synchronize every D units of virtual time on *appointments*, or synchronization points which ensure that neither timeline advances too far and creates causal error by receiving an event in the past. If the sending timeline reaches its appointment K before the receiving timeline, the sending timeline continues to advance. Otherwise, the receiver timeline reaches its appointment K and waits for

the sender timeline to also reach K.

2.4.1 S3F Performance

The performance of an S3F simulation depends entirely on the model and how entities are aligned. Entities synchronize efficiently when they avoid cross-timeline channels by being aligned on the same timeline. Whether a cross-timeline channel is *fast* or *slow* depends entirely on the global barrier ω. Consider two *extreme* cases for ω.

ω makes all cross-timeline channels *slow*: All the synchronization occurs at the global barrier ω. This is attractive when an entity, such as a router, is connected to a large number of other entities on other timelines. In this case, the cost of appointment synchronization is avoided since timelines will skip appointments and synchronize at ω.

ω makes all cross-timeline channels *fast*: All the synchronization occurs at appointments. This is attractive when there are only few small cross-timeline channel delays when compared with the average cross-timeline delay. Furthermore, this situation is well suited for tighter coupling of simulation and emulation as it allows timelines to advance by the smallest amount of virtual time during which an emulated entity could affect another emulated entity.

With a simulation consisting of purely simulated entities, it is possible to simulate 60 minutes of virtual time within a smaller time frame such as 60 seconds. Of course, this is not always the case and this also largely depends on the underlying hardware, simulation model, and the number of timelines. However, with emulation in the picture, an hour long simulation

is guaranteed to take at least an hour when LXCs are dilated with TDF ≥ 1. As the following sections explain, the runtime depends on the largest TDF of emulation.

2.5 Design Choices

S3FNet allows a modeler to specify the time unit of the simulation. A simulation second can have 1,000,000 ticks if the time unit is microseconds. Similarly, a simulation second can have 1,000,000,000 ticks indicating the time unit is nanoseconds. S3FNet-LXC focuses on simulations with a time unit of microseconds. This choice was made due to the nature of *gettimeofday(...)*. A *gettimeofday(...)* function returns to a calling process the wallclock time, in microseconds, elapsed since Epoch time. If a process is dilated, then *gettimeofday(...)* returns a process's virtual time instead of wallclock time. At the beginning of an S3FNet-LXC simulation, LXCs are created and frozen at time T_0. This virtual time is recorded. Next, when an LXC's virtual time is queried at $T_1 > T_0$, the difference, $(T_1 - T_0)$, is recorded, signifying the elapsed virtual time since the simulation began. This is useful as it allows S3FNet-LXC to work on 32-bit operating systems. Virtual time in S3FNet is represented by a signed long integer type. Since virtual time is recorded in microseconds, a timeline can have a maximum virtual time of 2,147,483,647 microseconds, or 35 minutes. Not an ideal solution, but it allows S3FNet-LXC to run on a 32-bit operating system as TimeKeeper was initially developed for a 32-bit environments. This limitation, of course, is essentially non-existent on 64-bit operating systems.

In order to simplify model design with emulation, S3FNet-LXC has the ability to specify emulation behavior from inside the DML file since main-

```
total_timeline 2
tick_per_second 6
run_time 10
seed 1
log_dir logOutputDirectory

Net [
    lxcConfig [
        settings [ lxcNHI 0:0 TDF 2.0 cmd "ping 10.10.0.10" ]
        settings [ lxcNHI 1:0 TDF 2.0 ]
    ]
    Net [
        id 0 alignment 0
        host [
            id 0 isEmulated 1
            graph [
            ProtocolSession [ name lxcemu use "s3f.os.lxcemu"  ]
            ProtocolSession [ name ip use "s3f.os.ip" ]
            ]
        ]
        interface [ id 0 _extends .dict.1Mb ]
    ]
    Net [
        id 0 alignment 0
        host [
            id 0 isEmulated 1
            graph [
            ProtocolSession [ name lxcemu use "s3f.os.lxcemu"  ]
            ProtocolSession [ name ip use "s3f.os.ip" ]
            ]
        ]
        interface [ id 0 _extends .dict.1Mb ]
    ]
    link [ attach 0:0(0) attach 1:0(0)
           min_delay 1e-6 prop_delay 0.001 ]
]
```

Figure 2.3: Sample LXC DML File Modeling 2 Emulated Hosts

taining and modifying large network models can be challenging.

2.6 Domain Modeling Language

Domain Modeling Language (DML) [2] is used to create concise and reusable network models for running experiments. In S3FNet, a DML file specifies a network topology, network behavior, simulation length, and number of

timelines. A sample DML file for the simplest S3FNet-LXC experiment is shown in Figure 2.3. In it, 2 emulated entities are aligned on 2 different timelines and connected with a link simulating a 1000 μs propagation delay. Furthermore, each LXC can be configured with a specific TDF and executable bash command.

2.7 LXCEmuSession

LXCEmuSession is a protocol session implemented in order to facilitate packets between individual entities in an S3F simulation with the help of *LXCEmuMessages*. It is similar to the OpenVZEmuEventSession in S3FNet-OpenVZ, however, a key difference is that when an *LXCEmuMessage* is propagated up the simulated protocol stack into *LXCEmuSession*, it is immediately injected into the *frozen* LXC associated with that *LXCEmuSession*. This ensures that a packet is delivered into an LXC as accurately as possible. This behavior is described in more detail in Section 2.10.

2.8 LXC Proxy

An LXC Proxy is a data structure that stands directly between the LXC and S3FNet-LXC. As shown in Figure 2.4, the LXC Proxy contains necessary information about both the LXC and the *ghost* host that represents it in the simulation. This *ghost* host behaves just like a host in a base S3FNet simulation except the traffic it pushes down its simulated network stack comes from an LXC. An LXC Proxy also maintains which TAP device the LXC is connected to. Through a TAP device, packets are written and read to and from processes via Linux file descriptors. An LXC Proxy must also keep

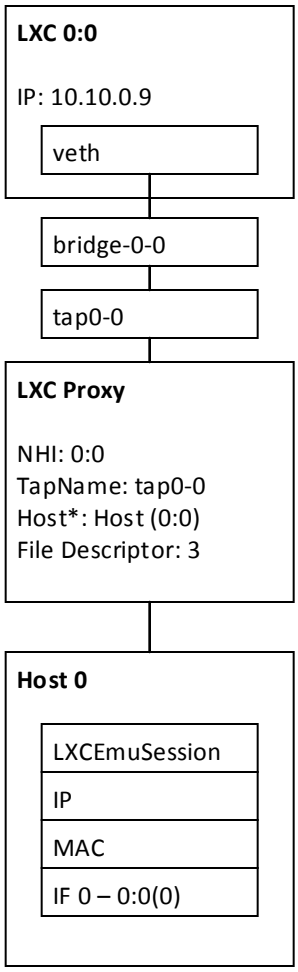

Figure 2.4: LXC Proxy Architecture

track of which simulated *ghost* host it is associated with as well as the LXC's parent PID. Figure 2.5 shows all the necessary components that acts as glue joining LXCs and S3FNet-LXC. These LXCs are created and maintained in the heart of S3FNet-LXC: the LXC Manager.

S3FNet's most common constructs are *Nets*, *Hosts*, and *Network Interfaces*. Nets can be composed of multiple nets (subnets), or multiple hosts. An S3F host must have at least one network interface. This host entity can be uniquely defined by concatenation of various nets that model it. For in-

stance, consider a net with ID = 5 that contains another subnet with ID = 4, which contains a host with ID = 200. S3FNet uniquely identifies this host as 4:5:200. This translation is then used to create and manage unique LXCs. However, Linux does not allow for the creation of bridges and TAP devices with names containing a colon; instead, the colon is substituted with a dash. If this host 4:5:200 was emulated, its corresponding LXC would be lxc4-5-200, the Linux bridge would be br-4-5-200, and the TAP device would be tap4-5-200.

```
unsigned int lxcIP; // IP Address associated with LXC
int fd;             // File Descriptor of TAP Device
Host* ptrToHost;    // Simulation ghost node
char tapName[100];  // TAP used with LXC Proxy
char lxcName[100];  // Name of LXC
char brName[100];   // Name of Linux Bridge interface
int PID;            // PID of LXC Parent process
double TDF;         // Time Dilation of LXC
```

Figure 2.5: Main Components Of An LXC Proxy

2.9 LXC Manager

S3FNet was expanded and modified in order to achieve integration with TimeKeeper. The S3FNet-LXC architecture is shown in Figure 2.6. A separate module-like library, LXC Manager, was created to work alongside S3FNet to facilitate the travel of packets in and out of LXCs. LXC Manager maintains a list of LXC Proxies which serve as liaisons between S3FNet and individual LXCs.

Consider the following scenario. LXC A with IP address IP_A has a proxy to a *ghost* Host H_A which is aligned on timeline TL_A. Similarly, LXC B with

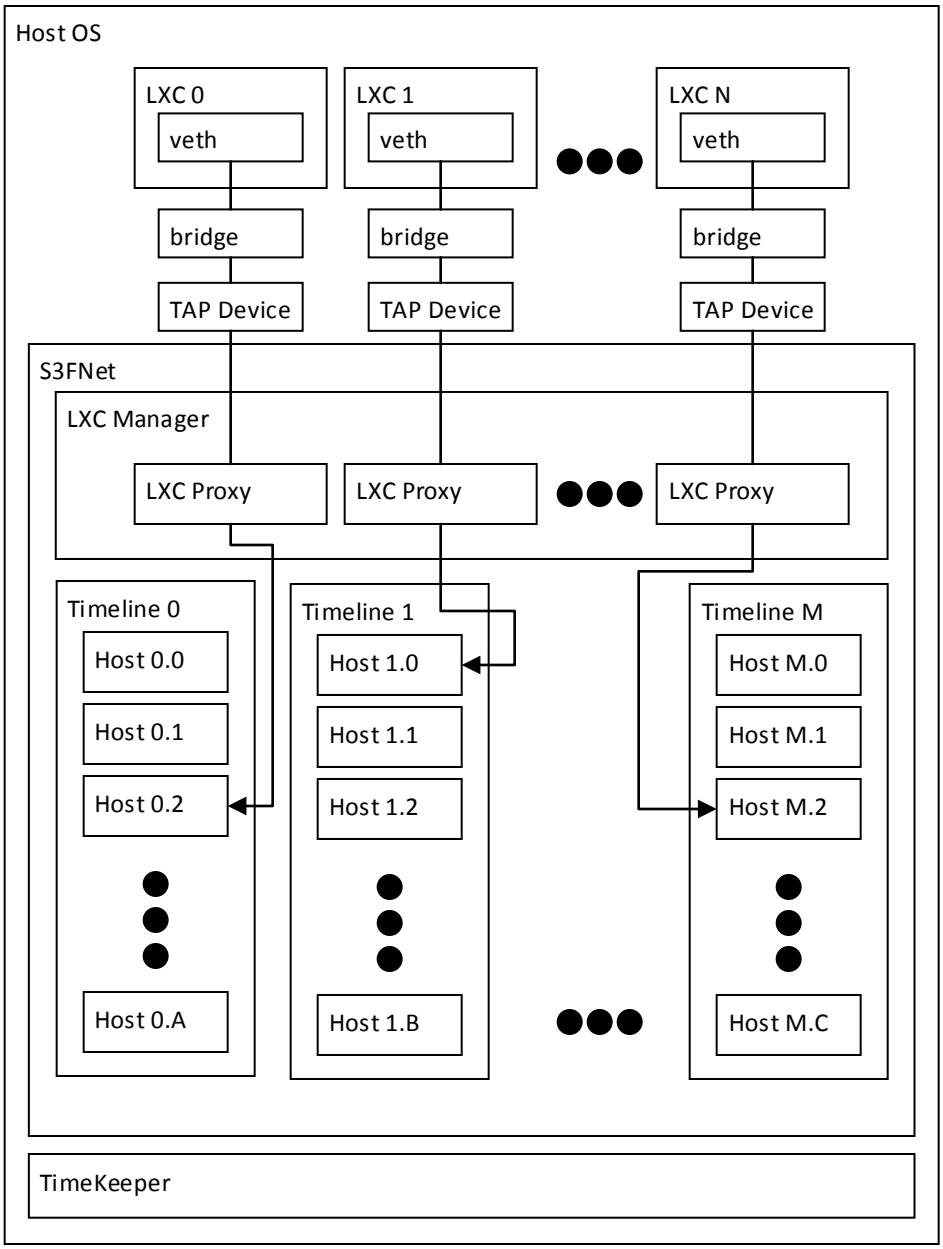

Figure 2.6: The Big Picture: S3FNet with LXCs and TimeKeeper

IP address IP_B has a proxy to a *ghost* Host H_B which is aligned on timeline TL_B. Suppose that LXC A wants to send a packet P to LXC B. The LXC Manager captures this packet, extracts the destination IP address (IP_B), and determines that the destination is LXC B. An LXCEmuMessage, destined

for H_B, is injected to the top of H_A's protocol stack. This makes it seem that packet P originated from H_A inside the simulation. Using S3F events, this packet gets propagated down H_A's protocol stack, through a cross-timeline channel, into H_B. Ultimately the packet will get injected into LXC B.

The LXC Manager has a single receiver thread which is responsible for capturing packets from LXCs. When an LXC sends out a packet, the receiver thread captures it and injects it into the appropriate *ghost* host inside the simulation. The LXC Manager determines the virtual timestamp of the packet by immediately querying the sending LXC for its virtual time. When an emulation event is processed inside the simulator, it emerges on the top most layer of a *ghost* host's protocol stack (*LxcEmuSession*). At that time, the packet is injected back into the LXC with the help of the LXC Proxy.

When the received thread in the LXC Manager captures a packet from an incoming LXC, it must determine which LXC it is coming from, and in most cases, the destination LXC of that packet. Since the LXC Manager keeps track of all LXC proxies, if a packet is destined for an LXC that is not in the simulation, then the packet is never injected into the simulator.

The LXC Manager has a crucial role in the advancement of LXCs in virtual time. Before timeline T advances its virtual time to J, the LXC Manager instructs the emulated entities aligned on T to advance their virtual clock and freeze at virtual time J. At time J, two scenarios can happen. At J, a packet is delivered to the frozen destination LXC or timelines simply synchronize at J and wait for the LXC Manager to advance LXCs again. See Figure 2.7.

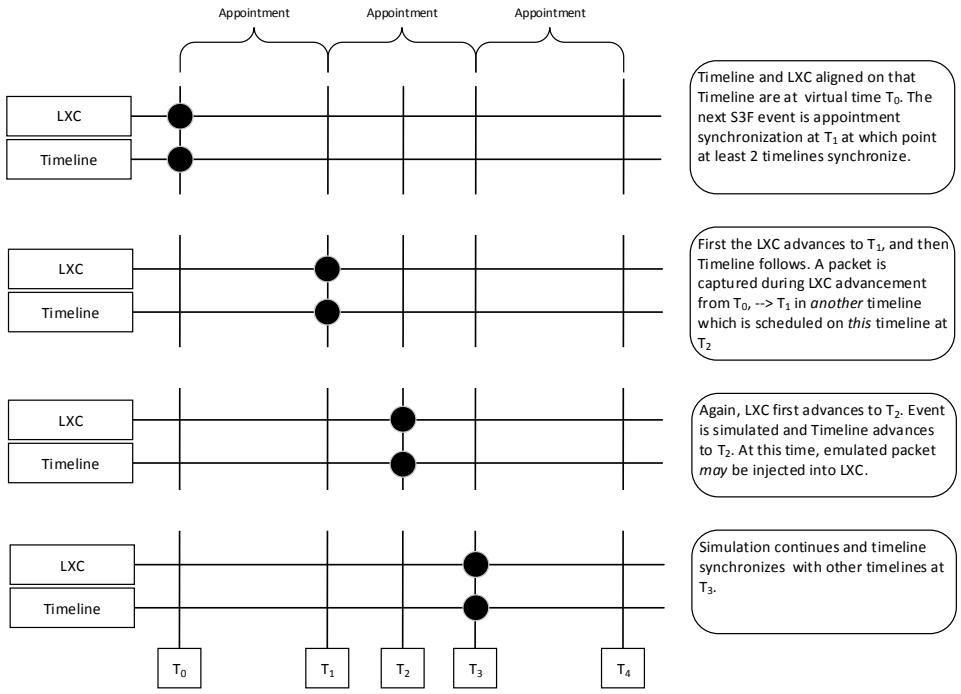

Figure 2.7: Close Coupling Between LXC And The Timeline On Which The LXC Is Aligned On

2.10 Modifications to S3F Kernel

The timeline class in the S3F API is the most crucial component of S3F. A timeline works in the space of a single *pthread* and is responsible for advancing its virtual time as S3F events are executed. In order to achieve integration with TimeKeeper, the timeline logic was slightly modified. However, S3F events which represent simulated and emulated packets can mingle among each other on the same timeline and that must be handled accordingly. Figure 2.8 shows the original timeline logic inside base S3FNet.

Consider the following general scenario where entities E_1, E_2 aligned on *different* timelines T_1, T_2, share a cross-timeline channel with delay D. If a timeline only models purely simulated entities, then it behaves as a regular timeline because no entities associate themselves with an LXC Proxy.

```
do forever
{
  get_events_from_other_timelines()
  calculate_synchronization_window()
  do
  {
    // execute synchronization window
    while (can_process_S3F_event_from_eventList())
    {
      E = next_event()
      T = E.get_time()
      advance_timeline_to(T)
      remove_event_from_eventList(E)
      process_and_simulate_event(E)
    }
  }
  while (timeline_not_reached_epoch_end())
}
```

Figure 2.8: Base S3FNet Timeline Logic

However, if a timeline models *at least* 1 entity that has a corresponding LXC Proxy, then its behavior slightly changes.

Now, assume that both E_1, E_2 simulate traffic which is generated by LXCs; this means that E_1, E_2 each associate themselves with a unique LXC Proxy. Both timelines and both LXCs start at virtual time π. To start the simulation off, both timelines calculate an appointment to meet at $(\pi + D)$. This is represented by *appointment* events with timestamps $(\pi + Di)$ for $(i = 1, 2, ...)$. Timelines begin concurrently processing events in their respective event lists which currently only contain appointment events. In the base S3FNet version, a timeline would *immediately* advance its virtual time to $(\pi + D)$ and *then* execute the appointment synchronization event.

With emulation, timelines must first instruct their LXCs to advance to $(\pi + D)$ before timelines can advance their own virtual time to $(\pi + D)$. Both LXCs are still frozen at virtual time π and both LXCs can generate

```
do forever
{
  get_events_from_other_timelines()
  calculate_synchronization_window()
  do
  {
    // execute synchronization window
    while (can_process_S3F_event_from_eventList())
    {
      E = next_event()
      T = E.get_time()
      if (timeline_models_at_least_1_LXC())
      {
        if (LXCs_not_at_time(T))
        {
          advance_LXCs_on_timeline_to(T)
          continue
        }
      }
      advance_timeline_to(T)
      remove_event_from_eventList(E)
      process_and_simulate_event(E)
    }
  }
  while (timeline_not_reached_epoch_end())
}
```

Figure 2.9: S3FNet-LXC Timeline Logic

packets at virtual time $[\pi, \pi + D]$ once they are unfrozen; these packets will become S3F events as they travel from one timeline to a *different* timeline *after* $\pi + D$. Note, these events need to be processed *before* the appointment events at $(\pi + D)$. If no event is generated during this time ($[\pi, \pi + D]$), then the simulation continues normally. However, suppose that LXC belonging to E_1 does generate a packet with a virtual timestamp $\beta = [\pi, \pi + D]$ meant for LXC on E_2. This packet will NOT affect the destination timeline T_2 until the next appointment. S3F achieves this by scheduling a special *timeout* event in T_1 with virtual timestamp $(\pi + \beta)$. This timeout event will be put into a

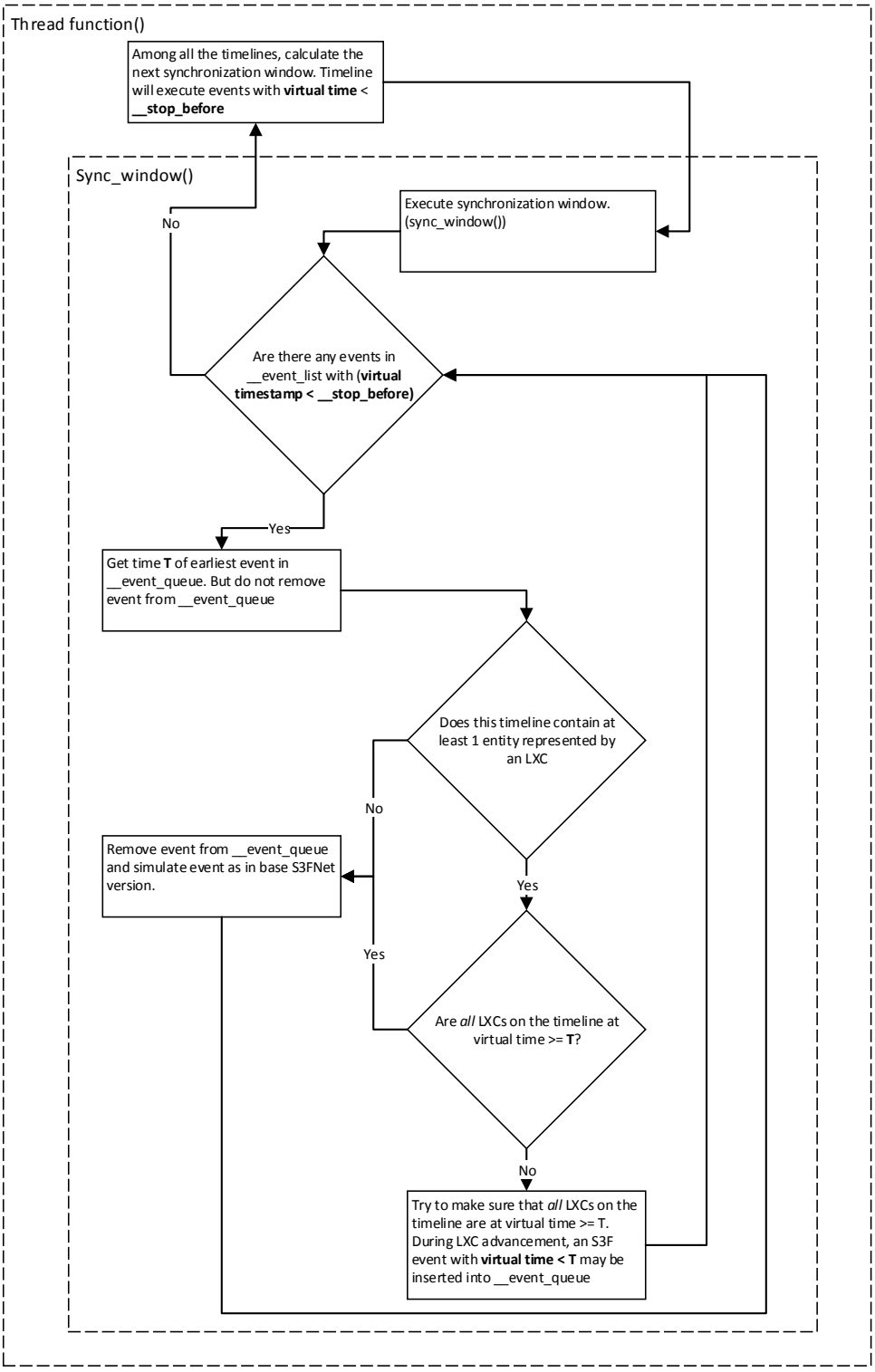

Figure 2.10: Modified Timeline Flowchart

T_1's event list (a priority queue), which ensures that the timeout event will be the next event executed by T_1, or, at the very least, it will be executed by T_1 before the appointment synchronization event.

When this timeout event is processed, T_1 will schedule an *activation* event J with virtual timestamp τ_J on T_2 where $\tau_J = (\pi + \beta + D)$ such that $(\pi + D) < \tau_J < (\pi + 2D)$. This activation event will ultimately propagate a message up the E_2's protocol stack. At that point, the protocol session decides what happens with that message.

Now, both timelines and LXCs have advanced to virtual time $(\pi + D)$. Their next appointment window is $[\pi + D, \pi + 2D]$. T_2 has to execute an activation event at $\tau_J = (\pi + \beta + D)$ that was generated in its previous appointment. This simulates a packet traveling across the wire with delay D. Recall that at this time, both LXCs and both timelines are synchronized at virtual time $(\pi + D)$. To ensure accurate injection into the LXC, the timeline instructs the LXC belonging to T_2 to advance by β virtual seconds and *freeze*. This will freeze the LXC at time τ_J. Next, S3FNet processes the event J and a *LxcEmuMessage* is propagated up the protocol stack to the *LxcEmuProtocolSession*. The receiving *ghost* entity has a corresponding LXC Proxy which contains information of the file descriptor utilized by the TAP device. At time τ_J, the packet generated at β is injected into the frozen LXC. Once the LXC unfreezes, the LXC processes said packet. Figure 2.9 showcases a flow chart summarizing the timeline logic above. Figure 2.10 shows the pseudocode illustrating the change from base S3FNet timeline to S3FNet-LXC timeline.

TDF	LXC Advancement Minimum T_{vt} (μs)
1	10
2	5
5	2
10	1

Table 2.1: Sweet spot for LXC advancement.

2.11 LXC Advancement Inaccuracy

The LXC Manager works together with TimeKeeper to advance LXCs in virtual time. As mentioned in Section 2.10, a timeline inside an S3F simulation may want to advance an LXC with TDF τ by an interval of virtual time T_{vt}. This tells TimeKeeper to advance that LXC by wallclock time T_{wc} where T_{wc} is T_{vt} multiplied by the LXC's TDF τ. In short, $T_{wc} = T_{vt} \times \tau$. However, TimeKeeper cannot accurately advance by extremely short intervals of wallclock time. Cases when T_{wc} is too small (less than 10 μs) must be handled separately.

The smallest possible interval to advance an LXC by is 1 μs. However, when $\tau = 1$ then $T_{wc} = 1$ and TimeKeeper will not accurately advance. Table 2.1 shows the minimum advancement of virtual time inside an LXC. This minimum threshold can vary depending on the underlying hardware. In order to avoid situations when an advance interval is too small, using a TDF ≥ 10 will suffice.

The question remains, how to handle the situation when the advance interval is too small? On one hand, the LXC can skip advancing T_{wc} and just increment its virtual time by T_{wc}. The LXC will not execute during this time. However, this creates a problem when a timeline attempts to advance an LXC by this small interval multiple times in series. Specifically, if a timeline consecutively advances an LXC by $T_{wc} < 10$ N times, then the

LXC will skip forward in wallclock time by $T_{wc} \times N$ without executing its processes. An alternative to this is to buffer the amount an LXC needs to execute by combining multiple T_{wc} values into a single advancement burst when the their wallclock time sum is greater than 10. This ensures that no execution time is lost and ensures that TimeKeeper does not have to advance LXCs by extremely small intervals. S3FNet-LXC implements the latter option.

2.12 LXC Advancement Algorithm

For the purposes of individual LXC management, TimeKeeper provides 5 key functions:

- *addToExperiment(PID, TimelineID)*: This notifies TimeKeeper which LXC will be part of a particular experiment. TimelineID is used to parallelize LXC advancement. Specifically, each S3F timeline will have a kernel worker thread responsible for advancing LXCs aligned on it. Recall that a timeline works in the space of a Linux *pthread* and will be responsible for advancing LXCs independent of other timelines.

- *synchronizeAndFreeze()*: This will freeze *all* the LXCs at the same virtual time. This virtual time acts as the starting point of the S3FNet-LXC simulation.

- *setInterval(PID, advanceTime, TimelineID)* This notifies TimeKeeper which LXC, and on which timeline, will want to advance by the interval advanceTime in virtual microseconds. No advancement occurs until *progress(...)* is called.

- *progress(TimelineID, FLAG)*: This notifies TimeKeeper to advance all

requested LXCs via *setInterval(...)*. When *progress(...)* is called, TimeKeeper wakes up a timeline's personal kernel worker thread which begins advancing LXCs. An optional flag can be set that will set the expected virtual time regardless of the actual virtual time an LXC advanced to.

- *reset(TimelineID)*: This notifies TimeKeeper to reset all LXC advancement configuration options requested by a specific timeline via *setInterval(...)*.

When a timeline η wants to advance its emulated entities, η asks the LXC Manager to advance the LXCs of η's emulated entities via the function *advanceLXCsOnTimeline(timelineID, timeToAdvanceTo)*. TimelineID specifies the timeline, and timeToAdvanceTo is the *absolute virtual time* the LXCs should advance to.

At this time, the LXC Manager inspects the LXCs η owns and determines which, if any, need to be advanced. If there are no LXCs associated with η, *advanceLXCsOnTimeline* returns false and η acts as a regular base S3FNet timeline. If there are LXCs associated with η, the LXC Manager individually queries η's *frozen* LXCs for their virtual time (*lxcTime*) and calculates the relative virtual time interval T_{vt} to advance by where T_{vt} = (*timeToAdvanceTo* - *lxcTime*). If an LXC with TDF τ needs to advance, and the interval ($T_{vt} \times \tau$) is greater than 10, then *setInterval(PID, T_{vt}, η)* is called with the relevent parameters including the LXC's parent PID. Once the LXC Manager established which LXCs will be advancing, *progress(η, FLAG)* is called. At that time, TimeKeeper begins to advance LXCs belonging to entities on η, while η waits. When TimeKeeper finishes advancing the LXCs, TimeKeeper notifies the LXC Manager via an inter process message

```
bool advanceLXCsOnTimeline(unsigned int timelineID,
                           ltime_t timeToAdvanceTo)
{
  if (timelineProxyList.empty())
    return false;

  int numberOfLXCsAdvancing = 0;

  for (each proxy P aligned on timelineID)
  {
    LxcAdvanceInterval = timeToAdvanceTo - P.VT
    if (LxcdvanceInterval <= 0) // no need to advance
      continue;

    if (LXCAdvanceInterval * P.TDF >= 10)
    {
      setInterval(P.PID, LxcAdvanceInterval, timeline);
      numberOfLXCsAdvancing++;
    }
    else
    {
      // do not advance LXC
      accumulateLXCTime
    }
  }

  if (numberOfLXCsAdvancing == 0)
    return false;

  // at least 1 LXC will advance
  progress(timelineID);
  // calculate individual LXC advance errors
  reset(timelineID);
  return true;
}
```

Figure 2.11: Algorithm Used By LXC Manager To Advance LXCs On A Given Timeline

and control returns back to η. At this point, *advanceLXCsOnTimeline(...)* calls *reset(TimelineID)* to reset LXC advancement intervals as they may be different the next time a timeline advances its LXCs. *advanceLXCsOnTimeline(...)* returns true if *at least* one LXC advanced in virtual time. Otherwise, it returns false signifying that no LXCs were advanced.

When *advanceLXCsOnTimeline(...)* finishes, η can process an S3F event or advance LXCs that did not reach the expected virtual time. If some LXCs undershot and advanced by less virtual time than expected, then the LXC Manager instructs TimeKeeper to advance these LXCs by the remaining interval of virtual time. Note, this interval is *at most* 1-2 virtual μs.

2.13 LXC Packet Timestamps

The accuracy of the simulation largely depends on the accuracy of LXC-generated packet timestamps. If an LXC sends out a packet with timestamp T_1, then the packet should be simulated in the simulator at T_1. This, however, requires for the LXC Manager to figure out the timestamp of an LXC on *microsecond* accuracy. Currently, this is accomplished by an LXC *timestamp query*. A timestamp query simply asks the LXC for its current virtual time. This query happens as soon as the LXC Manager notices that a particular LXC's TAP device was modified. If a packet is sent via an LXC's virtual Ethernet interface at virtual time T_1, that LXC's virtual time will be queried by the LXC Manager at virtual time $T_1 + \epsilon$. The sequence of events that occur is showcased in Figure 2.12. This timestamp error ϵ varies with an LXC's TDF and it is possible for ϵ to be *zero*. Recall that the higher the TDF of an LXC, the longer it will take in *wallclock* time to advance an LXC by a *single* virtual second. For instance, if an LXC is dilated with TDF

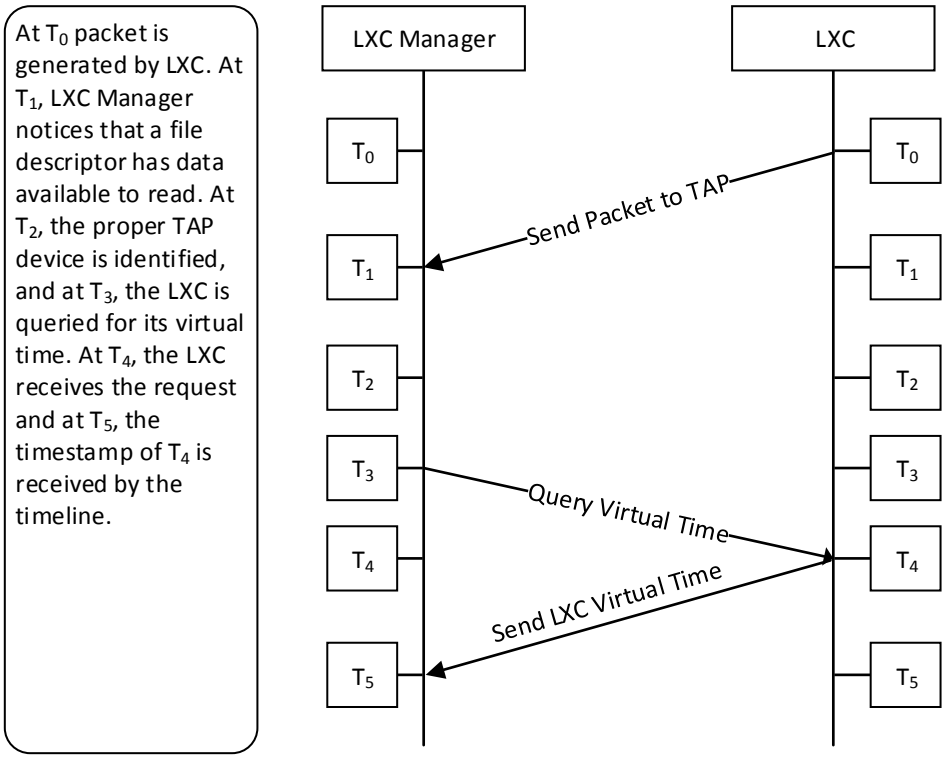

Figure 2.12: Determining Timestamp of LXC Generated packet.

200, then the LXC Manager will have *at most* 200 wallclock microseconds to query an LXC's virtual time before it advances by a virtual microsecond. Of course, at the time of query, the LXC may have already advanced by, say, 96 wallclock microseconds and there are only $(200 - 96) = 104$ wallclock microseconds before an LXC's virtual time increments by 1 microsecond. It is possible to query a packet timestamp with the correct virtual time, as long as the LXC is queried within the *exact virtual microsecond* of the packet's delivery to the TAP device.

CHAPTER 3

RESULTS

This chapter contains the evaluation and analysis of the integration of S3FNet with TimeKeeper. The particular focus is feasibility, practicality, accuracy, and repeatability with various types of experiments.

3.1 Testing Environment

All experiments were conducted using a modified 3.10.9 64-bit Linux kernel which implements TimeKeeper. Experiments 1-5 were conducted on a desktop consisting of 8 Intel Core i7-2600s @3.40 GHz and 16 GB of RAM. In order to measure and test scalability, experiments 6-7 were conducted an enterprise server consisting of 24 Intel Xeon X5650s @2.67 GHz and 24 GB of RAM.

3.2 Experiment 1: Simulation Accuracy and Correctness

Experiment 1 and 2 simulates a model of 2 LXCs aligned on 2 different timelines "pinging" between each other 3000 times. However, this is not the standard Linux ping program due to ping's current restriction to wait a minimum of 1 wallclock second before pinging again. Instead a pseudo ping is defined as follows. One LXC acts as a UDP server and the other LXC

TDF	Avg Ping RTT (μs)	Std Dev Ping RTT (μs)	Max Advance Error (μs)	Avg Advance Error (μs)	Std Dev Advance Error (μs)	95 % Confidence Interval (μs)
1	704.57	11.52	19	3.74	2.77	0.41
2	663.41	5.01	12	2.06	1.26	0.18
3	654.86	5.44	10	1.68	1.06	0.19
4	649.84	2.78	9	0.72	1.28	0.1
5	647.03	2.52	6	0.59	0.55	0.09
6	645.58	2.34	3	0.74	0.46	0.08
7	644.48	2.18	3	0.73	0.46	0.08
8	643.64	2.16	4	0.84	0.38	0.08
9	642.84	2.04	3	0.86	0.36	0.07
10	642.21	1.9	3	0.11	0.32	0.07
11	641.87	1.97	2	0.1	0.3	0.07
12	641.13	1.74	2	0.08	0.27	0.06
13	639.84	1.15	2	0.05	0.22	0.04
14	640.9	1.84	2	0.03	0.16	0.07
15	640.29	1.65	2	0.05	0.22	0.06
16	640	1.73	5	0.03	0.17	0.06
17	639.88	1.55	2	0.03	0.17	0.06
18	639.57	1.49	2	0.02	0.15	0.05
19	639.47	1.44	2	0.01	0.08	0.05
20	638.86	1.49	1	0.01	0.09	0.05
25	638.34	1.27	1	0	0.05	0.05
30	637.65	1.17	3	0	0.02	0.04
35	636.87	0.76	1	0	0.01	0.03
40	636.7	1.11	1	0	0.01	0.04
45	636.52	0.79	0	0	0	0.03
50	635.95	0.77	0	0	0	0.03
100	634.54	0.68	0	0	0	0.02

Table 3.1: Accuracy of simulation with varying TDFs among 3000 pings.

the acts as a UDP client. At virtual time V_1, the UDP client sends out a message M_1 to the UDP server. As soon as the UDP server receives M_1, it immediately replies with M_2 to the UDP client. When the UDP client receives M_2 at virtual time V_2 the difference, $(V_2 - V_1)$, is recorded as the pseudo ping RTT. In this experiment, the 2 simulated hosts have a 100 μs channel delay between them. A ping RTT consists of 2 packets of equal size, each with a transmission time of 217 μs. Therefore, the perfect ping time between the 2 LXCs should be $100 \cdot 2 + 217 \cdot 2$ which adds up to 634 μs.

Table 3.1 shows how the accuracy of the ping RTT changes when LXCs are dilated with different TDFs. When TDF increases, the overall RTT error is expected to decrease because packet timestamps are more accurate. Additionally, the error during LXC advancement is reduced because it takes a longer amount of wallclock time to increment virtual time. With smaller TDFs, the decrease in ping RTT error is more visible. With a TDF of 10, the ping RTT comes within 8 μs of the desired 634 μs result. With a TDF of 20, the ping RTT comes within 4 μs of the desired result. However, the improvement in accuracy is more apparent between TDFs 1-20 than TDFs 20-50. With TDF 100, the ping RTT is almost identical (634.54 μs) to the expected ping RTT (634 μs). The standard deviation is very low which indicates that almost every single ping RTT is close to the expected value. Lastly, the confidence interval for each run shows that as the TDF increases, the margin of error decreases.

Between TDF 1 and 100, there is roughly a 70 μs difference between the average ping RTTs. A ping RTT consists of 2 packets which means that on average, there is a 35 error μs incurred per packet. This occurs because of the difference between the time when a packet is sent out via an LXC's virtual network interface and when the same packet is captured by the LXC

Manager.

Consider the following explanation for the high ping RTT when TDF is 1. Before the UDP client first sends out a packet, it calls *gettimeofday(...)* and records that virtual time V_1 as the virtual send time. Immediately, the UDP client sends a packet P_1 (ping request) to the UDP server. P_1 is captured by the LXC Manager which then queries the sending LXC for its timestamp. This virtual time is measured as $(V_1 + \epsilon_1)$ where $\epsilon_1 \approx 16$. ϵ_1 signifies the time taken for P_1 to travel from the LXC's virtual Ethernet across the Linux bridge into the TAP device until P_2's timestamp is ultimately queried by the LXC Manager.

Next, this packet is processed through the simulator and injected into the *frozen* LXC executing the UDP server at virtual time $(V_1 + \epsilon_1 + D)$ where D is the channel delay (100) plus the transmission delay (217). The LXC containing the UDP server is *unfrozen*. The UDP server receives this packet at $(V_1 + \epsilon_1 + D + \epsilon_2)$. As soon as the UDP server receives P_1, it calls *gettimeofday(...)* in order to measure that $\epsilon_2 \approx 16$ μs. ϵ_2 is the time taken for the injected P_1 to be seen inside the UDP server.

Immediately, the server sends a packet P_2 (ping reply) with an almost identical timestamp $(V_1 + \epsilon_1 + D + \epsilon_2 + \omega)$ where ω is the time taken by the UDP server to create, process, and send P_2. Again, the LXC Manager queries the LXC running the UDP server and learns that P_2's timestamp is $(V_1 + \epsilon_1 + D + \epsilon_2 + \omega + \epsilon_3)$. Since this the same as when the UDP client first sends P_1, $\epsilon_3 \approx 16$ μs. Like ϵ_1, ϵ_3 signifies the time taken for P_2 to travel from the LXC's virtual Ethernet across the Linux bridge into the TAP device until P_2's timestamp is ultimately queried by the LXC Manager. Once again, P_2 is processed through the simulator and injected into the LXC containing the UDP client at $(V_1 + \epsilon_1 + D + \epsilon_2 + \omega + \epsilon_3 + D)$. The UDP

client calls *gettimeofday(...)* and sees a virtual timestamp of $(V_1 + \epsilon_1 + D + \epsilon_2 + \omega + \epsilon_3 + D + \epsilon_4)$. Like ϵ_2, ϵ_4 is the time taken for the injected P_2 to be seen inside the UDP *client*. Next, the UDP client records the difference $[(V_1 + \epsilon_1 + D + \epsilon_2 + \omega + \epsilon_3 + D + \epsilon_4) - V_1] = (\epsilon_1 + D + \epsilon_2 + \omega + \epsilon_3 + D + \epsilon_4)$ as the ping RTT. Since $D = 317$, this means that $(\epsilon_1 + \epsilon_2 + \omega + \epsilon_3 + \epsilon_4) \approx 70$ μs. With TDF 1, on average $\epsilon_1 \approx \epsilon_2 \approx \epsilon_3 \approx \epsilon_4 \approx 16$ μs and ω is very small relative to ϵ. As the TDF increases, ϵ decreases because the LXC Manager has more time to query a packet for an accurate virtual time. Similarly, when a packet is injected into the LXC at T_1 and *the LXC* queries its timestamp at T_2, the difference $(T_2 - T_1)$ decreases to 0. Overall, as TDF increases $(\epsilon_1 + \epsilon_2 + \epsilon_3 + \omega + \epsilon_4 \to 0)$.

It is important to note, however, that throughout the simulation, one LXC will occasionally send ARP requests to the other LXC thereby creating extra traffic. This has the potential to affect the simulation as ARP packets injected into the simulation may slightly delay the arrival of a ping request or a ping reply packet. Consequently, this will make the ping RTT slightly higher than expected. These ARP requests are scheduled on a *wallclock time* basis. For instance, a 1 virtual second simulation with TDF 100 lasts *at least* 100 wallclock seconds. More occasional ARP requests will be sent in a simulation that lasts for a longer duration of wallclock time. As a result, these very few ping RTT anomalies were removed from the data presented in Table 3.1.

The main takeaway from this experiment is the idea that as the TDF increases, packet timestamps are more accurate, which creates more accurate emulations. These packet timestamps are more accurate because the longer it takes an LXC to advance by 1 virtual second, the more time the LXC Manager has to query the most accurate microsecond timestamp. This behavior

was discussed in Section 2.13. When TDF is 100, the expected and actual ping RTT are almost identical, nevertheless, there is still some uncontrollable behavior incurred by the Linux operating system.

3.3 Experiment 2: LXC Advancement Accuracy and Correctness

Table 3.1 also shows the advancement error of LXCs which is calculated on an LXC basis. After *progress(...)* returns, each LXC's actual virtual time is queried and compared with the expected virtual time. Their difference is recorded as the advance error. The longer the advance interval of an LXC, the more accurately TimeKeeper can advance it. Consequently, the largest error occurs when TDF is 1 due to the smallest advancements in wallclock time, i.e. an advancement of 10 virtual seconds is equivalent to an advancement of 10 wallclock seconds. As the TDF increases, the minimum wallclock advancement interval increases. After TDF 10, the *maximum* advance error is still greater than 1 μs, however, the average error is consistently smaller than 1 μs. Again, this increase in accuracy is largely due to the different wallclock time representations of virtual time. Overall, TimeKeeper does a better job of LXC advancement with higher TDFs.

TimeKeeper has the option of forcing the correct virtual time, even though the LXC advanced too far or not far enough, by passing a flag in *progress(...)*. In other words, an LXC wants to advance by K units of virtual time but instead advances by $(K \pm N)$ units of virtual time. TimeKeeper can force that LXC's virtual time to be K. This, however, fixes one error but introduces another type of error. For instance, an LXC advances incorrectly to $(K + N)$ and sends out a ping request at time $(K + N)$. TimeKeeper

forces the LXC's virtual time to be K. Eventually, a ping reply arrives but the ping RTT will be incorrectly off by N.

The main takeaway from this experiment is that the accuracy of LXC advancement also has a significant impact on the simulation. Smaller TDFs mean smaller intervals of wallclock time which means that LXCs will overshoot or undershoot their advancement with more error. As the TDF increases, LXCs advance more accurately. This also has the added effect of improving LXC packet timestamp accuracy as the LXC Manager has more wallclock time to query an LXC for its virtual time.

3.4 Experiment 3: Repeatability

Experiment 3 models the same network topology as experiment 1 and 2 but focuses on the repeatability and consistency of the emulated behavior. Specifically, this experiment focuses on repeatability by scaling the TDF to isolate and analyze the variance between runs caused by the non-determinism of TimeKeeper.

In this simulation scenario, the ideal message RTT would be 2434 μs. The experiment measures variance by running the same simulation with the same parameters multiple times and examining the difference of the i^{th} ping RTT between runs. In other words, to ensure consistent behavior, the RTT of ping SEQ number i should be close across all runs. The general trend is that as the TDF increases, the repeatability of a simulation improves. Figure 3.1 confirms this hypothesis, showing that with TDF 15, the average RTT of each ping is roughly 2444 μs. As the TDF increases to 30 and 45, the average ping RTT grows closer to the expected result while still remaining consistent. Figure 3.2 further supports this by showing that the RTT standard deviation

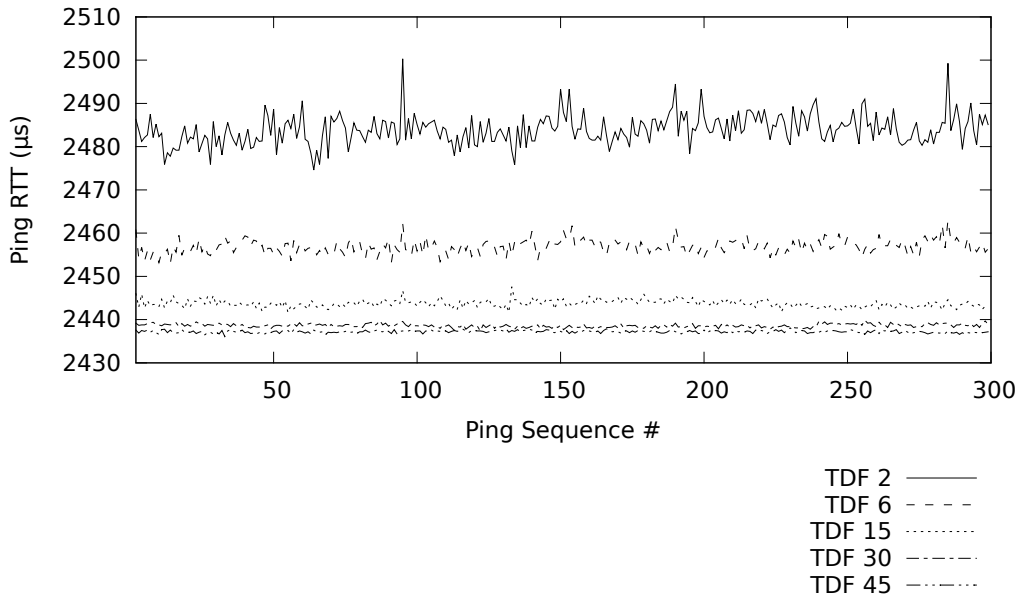

Figure 3.1: Average RTT With Emulated UDP Traffic

is consistent as well, hovering around 3 μs.

This experiment also measures the repeatability with TDFs smaller than 10. During such a simulation, it is possible to have situations where an LXC will not advance due to its need to advance by too small of an interval (see Section 2.11). Figure 3.2 shows that with such low TDFs, the standard deviation of ping RTT greatly fluctuates when compared to simulations with larger TDFs. This is expected as packet timestamps are the most inaccurate.

The main takeaway from this experiment is that despite TimeKeeper's non-determinism, simulations can more consistent depending on the TDF of LXCs. The higher the TDF, the more consistent the packet timestamps which means more consistent simulations. It would be difficult to repeatedly replicate perfect pings in a real world network. This tiny variance of the ping RTT can make the model's behavior similar to the behavior of real world networks.

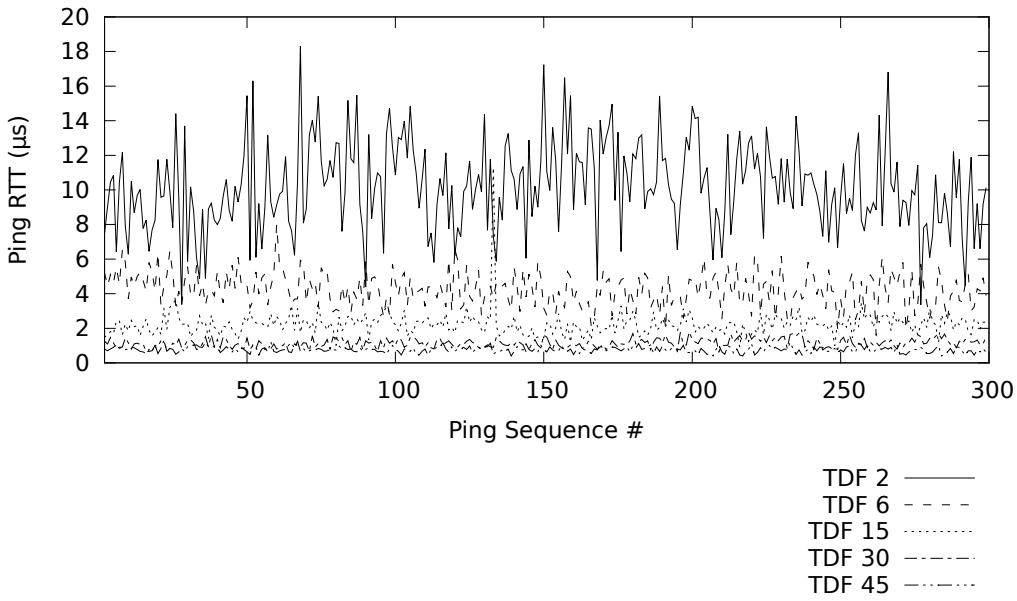

Figure 3.2: Standard Deviation RTT with Emulated UDP Traffic

3.5 Experiment 4: Mixed Traffic

Consider the following question: what happens when emulated and simulated hosts interact with one another? Experiment 4 seeks to explore that behavior. This experiment models 4 total hosts: 2 simulated hosts modeling TCP traffic where a TCP client deterministically downloads a file from a TCP server and 2 emulated hosts just like in experiment 1. However, this time, they share a link which means simulated and emulated traffic can affect each other. This scenario is shown Figure 3.3 where the 2 types of traffic going across link C potentially affect each other.

Like experiment 3, experiment 4 analyzes variance of emulation but also measures how repeatable a hybrid simulation can be. Specifically, it helps measure how accurately the i^{th} ping packet is received at virtual time J. Ideally, the i^{th} ping packet should be always received at the same virtual time J across repeated simulations. Emulated and simulated traffic clashes with each other as soon as the simulated TCP client starts downloading a

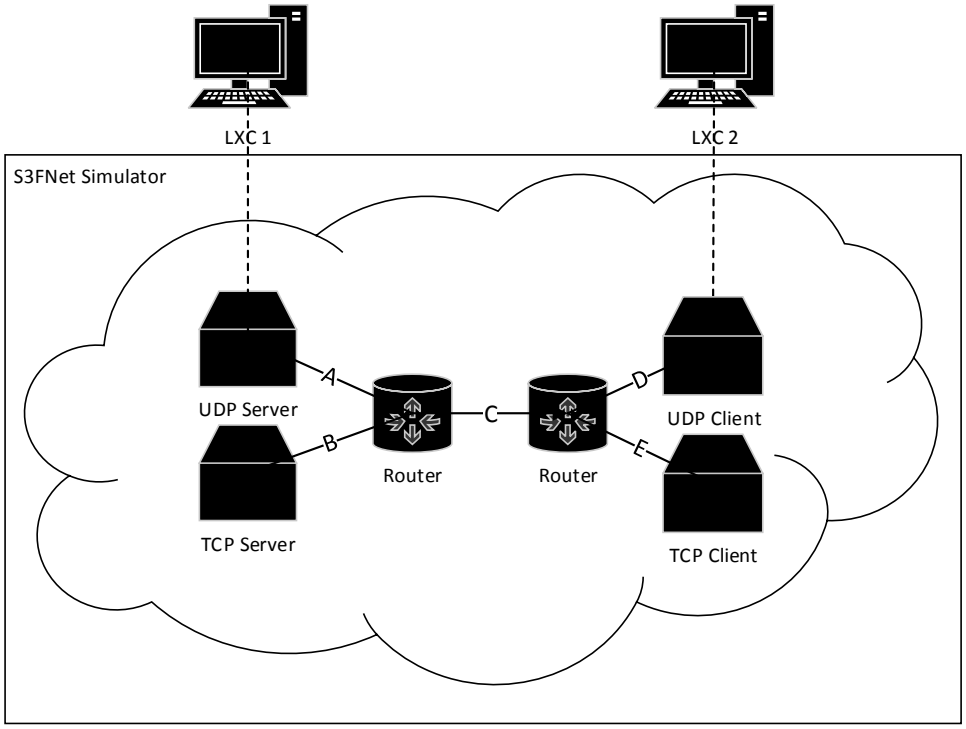

Figure 3.3: Simulation Model With Mix of Simulated And Emulated hosts

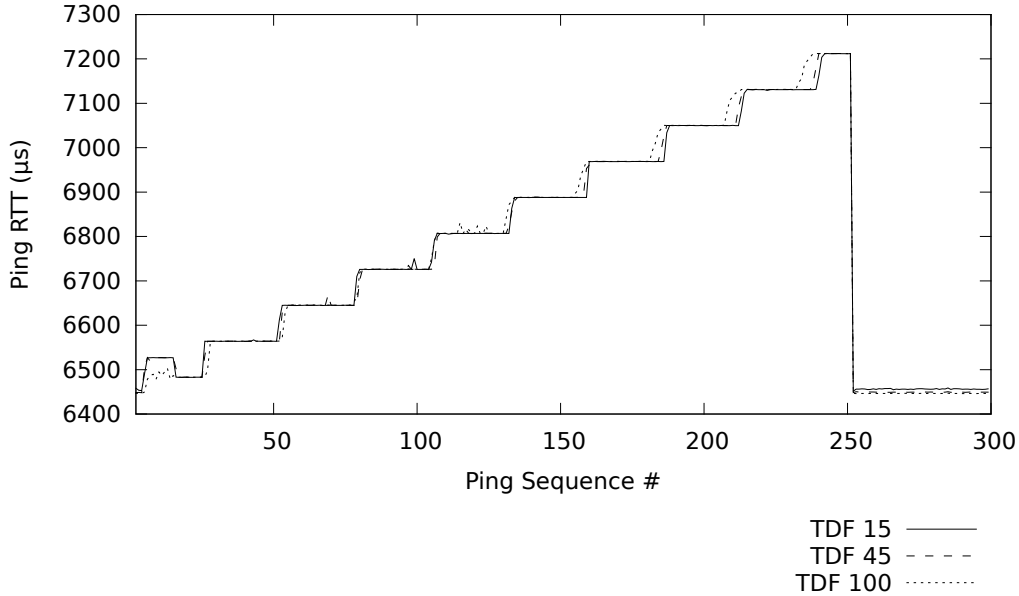

Figure 3.4: Average RTT with Emulated UDP Traffic

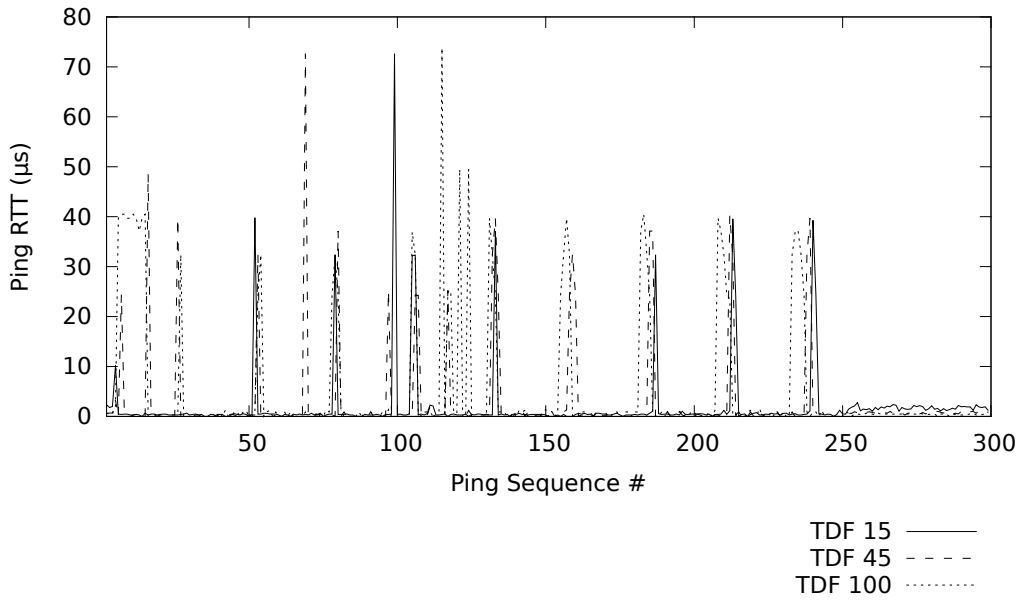

Figure 3.5: Standard Deviation RTT with Emulated UDP Traffic

file. Figure 3.4 showcases the average ping RTT as the UDP client pings the UDP host. This simulation is run multiple times with the LXCs dilated with TDF 15, 45 or 100. The average ping RTT increases because the TCP window size increases as the file is being downloaded. The emulated behavior is generally consistent, however, Figure 3.5 shows that there are more peaks with higher standard deviation. The TCP client finishes downloading the file at about the 250^{th} ping and the behavior normalizes.

This, albeit small standard deviation, is unavoidable due to the non-determinism of CPU execution. Since the simulated TCP traffic is deterministic, an emulated i^{th} ping RTT that arrives to the client at virtual time T_0 in one run, may arrive at virtual time $T_0 + \epsilon$ in another run even though ϵ is small. This inadvertently affects the next ping as emulated traffic collides with the TCP traffic at different moments in virtual time.

The main takeaway from this experiment is that it is very unlikely to replicate 2 instances of combined simulation and emulation due to the non-

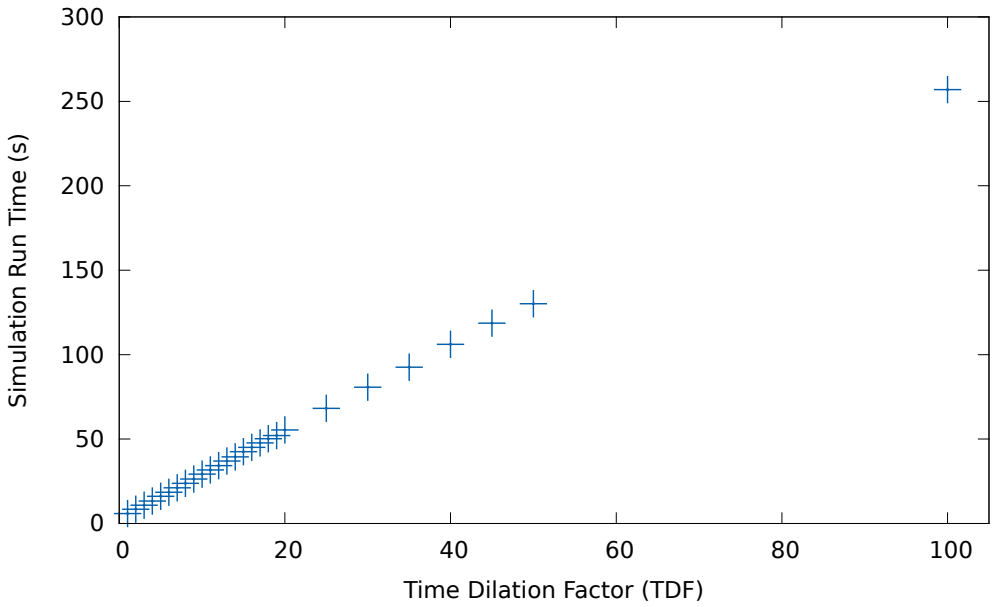

Figure 3.6: Run Time of Emulation with varying TDFs

determinism of the CPU scheduler. It is also difficult to consistently replicate when and how many machine instructions are actually executed. This depends on the precision of the Linux timer as well as other factors such as cache misses and page faults. Over multiple simulations of the mixed model, it is plausible to recreate the results with small error.

3.6 Experiment 5: Scalability

Performance improvement was the main driving force for the creation of SSF and consequently S3F. Wallclock time spent during LXC advancement, however, is the lowest bound on the runtime of an S3FNet-LXC simulation. In other words, a simulation that spends N wallclock seconds advancing LXCs, will take at least N wallclock seconds.

TimeKeeper allows the modeler to specify how many CPU cores (λ) are allocated for TimeKeeper use. TimeKeeper commandeers these CPUs in

order to advance processes on its own terms. In order to precisely run an LXC for an interval of I units of virtual time, there can be no preemption on the CPU during execution. Once a CPU unfreezes an LXC L with a TDF of τ, the same CPU will only immediately freeze L after L has executed on the CPU for $(I \times \tau)$ units of wallclock time. Disallowing preemption and context switches increases the accuracy of CPU time given to LXC L. Since a timeline can contain multiple LXCs, LXCs are allocated to each core on a timeline basis. Specifically, each LXC on timeline $i = 0, 1, 2...W$, is assigned to core $c = (i \mod \lambda) + 1$ with 1-based indexing. In more concrete terms, if an experiment models 12 timelines (with IDs 0-11), and allocates 3 cores for TimeKeeper use, core 1 will manage timelines 0, 3, 6, 9, core 2 will manage timelines 1, 4, 7, 10, and core 3 will manage timelines 2, 5, 8, 11.

Total amount of time spent advancing N virtual seconds of a single LXC dilated with TDF τ will take at *at least* N wallclock seconds assuming $\tau \geq 1$. Specifically, the lowest bound is $(\tau \times N)$. However, if a maximum number of K LXCs are aligned on a single timeline, then the lowest bound ρ increases to $\rho = (\tau \times N \times K)$ since TimeKeeper advances LXCs, aligned on a single timeline, in series. In other words, if a timeline wants to advance LXCs A, B, C, then the CPU core on which the LXCs are aligned on will first advance A, then B, and then C.

If the number of timelines in a simulation is less than or equal to the number of cores dedicated to it, then the emulation will take at least ρ seconds. Figure 3.6 showcases the run time of a fixed simulation of 2.5 seconds. With TDF 1, the total simulation run time is 5.88 wallclock seconds. Roughly 2.5 wallclock seconds of this time is spent advancing emulation, and the rest is incurred by the simulation overhead O. Thus $2.5 + O = 5.87$, where $O \approx 3.32$ wallclock seconds. With TDF 2, the same runtime

Cores Dedicated (λ)	Total Simulation Run Time T_{vt} (s)
1	1202.83
2	602.633
3	602.466
4	302.200
5	302.714
6	303.165

Table 3.2: *Total* Simulation Run-Time with Varying amount of cores

is 8.42 wallclock seconds, increasing by roughly 2.5 wallclock seconds. In other words, when TDF is 2, each timeline spends $2.5 \cdot 2$ wallclock seconds in emulation. Moreover, $(2.5 \cdot 2 + O = 5 + 3.32) = 8.32 \approx 8.42$. This trend increases linearly which means that the simulation runtime increases at a fixed cost of emulation.

For most optimal performance, each timeline would ideally contain a single emulated host. However, the physical limitation is the number of cores on a single machine. Thus, the next step up is to design a simulation model such that LXCs are equally placed among N timelines where N is the number of available cores on a host machine minus 2 (for background work). This means that each timeline has its own kernel thread responsible for advancing the LXC it owns. A simulation model, can of course, contain more than N timelines. Experiment 5 models a modest number of 100 LXCs where 25 LXCs are each aligned on 4 timelines. Table 3.2 shows the results of a 1 virtual second simulation.

- With 4 or more CPU cores dedicated to this experiment, each core will be responsible for advancing 25 LXCs concurrently. With a TDF of 12, the total simulation runtime of a single virtual second is expected to take at least $(12 \times 25 \times 1 = 300)$ wallclock seconds, as that is how much total wallclock time will be used to advance LXCs on a *single* timeline.

- With 3 CPU cores allocated to this experiment, core 1 will be responsible for advancing 50 LXCs aligned on timelines 0 and 3. Consequently, the lowest bound on the simulation run time is $(12 \times 25 \times 2) = 600$ wallclock seconds. The other 2 cores (core 1 and 2), will be responsible for each advancing 25 LXCs and will be waiting a majority of time for LXCs on timelines 0 and 3 to advance during timeline synchronization. The same behavior occurs when TimeKeeper utilizes 2 cores.

- With 2 CPU cores allocated to this experiment, both core 1 and 2 will be responsible for advancing 50 LXCs aligned on timelines (0,2) and (1,3) respectively. Again, the lower bound on the simulation run time is $(12 \times 25 \times 2) = 600$ wallclock seconds.

- With 1 CPU core allocated to this experiment, core 1 will be responsible for advancing *all* 100 LXCs. In a given *progress(...)* call, every LXC will be advanced sequentially and *progress(...)* will take at least $(12 \times 25 \times 4) = 1200$ wallclock seconds.

The main takeaway from this experiment is that LXCs need to be executed on a CPU core without context switching. The number of LXCs advanced concurrently depends on the number of CPU cores available. The more cores dedicated to an experiment, the more emulation parallelization can be exploited. Experiment 6 will further explore scalability.

3.7 Experiment 6: Scalability Continued - Thousands of LXCs

Another point of interest is the overhead during the advancement of LXCs. Specifically, all emulation happens when the LXC manager calls *progress(...)*

after configuring which LXCs will be advanced. Inside a single *progress(...)*, TimeKeeper may instruct hundreds of LXCs to advance by an interval of virtual time, though in this experiment that interval is *never* smaller than 900 virtual seconds.

Experiment 6 models thousands of LXCs on an enterprise server to measure scalability on a massive scale. Throughout this experiment, *no traffic* is sent between the LXCs and there are enough CPU cores allocated such that each timeline gets its *own* core. In the first pair of tests, 1000 LXCs are modeled on 4 timelines and simulate both 0.1 and 1 virtual second. Table 3.3 shows results from an experiment with 250 LXCs each aligned on 4 timelines with TDF 20.

The first set of columns shows how much wallclock time each timeline spends in advancing LXCs using the *progress(...)* function call during a simulation of 0.1 virtual seconds. This is measured by $(T_2 - T_1)$; T_1 is recorded just before *progress(...)* is called and T_2 is recorded as soon as *progress(...)* returns. The second set of columns shows the same statistics for a simulation of 1 virtual second. Each timeline manages 250 LXCs and TimeKeeper will spend $(0.1 \times 20) = 2$ wallclock seconds advancing an LXC through 0.1 seconds of virtual time. Therefore, 250 LXCs will need to individually advance by a total of $(0.1 \times 20 \times 250) = 500$ wallclock seconds.

The overhead measured is the time taken between interactions of the LXC Manager and TimeKeeper during the advancement of LXCs. When a timeline calls *progress(...)*, the LXC Manager sends a message to TimeKeeper, which in turn schedules LXCs on CPUs. Once all designated LXCs finished advancing, TimeKeeper must then notify the LXC Manager (and consequently the timeline) that *progress(...)* has finished. This is the overhead incurred when S3FNet-LXC uses TimeKeeper to advance LXCs.

	0.1 Second Simulation			1 Second Simulation		
TL	Time Spent in Progress (s)	Emulation Time (s)	Progress Overhead (%)	Time Spent in Progress (s)	Emulation Time (s)	Progress Overhead (%)
0	500.66	500	0.132	5006.45	5000	0.129
1	501.16	500	0.231	5011.32	5000	0.226
2	500.66	500	0.132	5006.86	5000	0.137
3	501.16	500	0.231	5010.93	5000	0.218

Table 3.3: 250 LXCs Aligned on 4 Timelines (1000 LXCs)

	0.1 Second Simulation			1 Second Simulation		
TL	Time Spent in Progress (s)	Emulation Time (s)	Progress Overhead (%)	Time Spent in Progress (s)	Emulation Time (s)	Progress Overhead (%)
0	800.97	800	0.121	8009.94	8000	0.124
1	801.08	800	0.135	8011.64	8000	0.145
2	801.04	800	0.13	8010.29	8000	0.128
3	801.09	800	0.136	8012.63	8000	0.158
4	801.02	800	0.127	8010.18	8000	0.127
5	801	800	0.125	8011.36	8000	0.142
6	800.98	800	0.122	8010.16	8000	0.127
7	801	800	0.125	8012.78	8000	0.159
8	801.41	800	0.176	8010.39	8000	0.13
9	801.07	800	0.134	8013.34	8000	0.166
10	801.06	800	0.132	8010.73	8000	0.134
11	801.04	800	0.13	8013.64	8000	0.17
12	801.04	800	0.13	8010.26	8000	0.128
13	801.1	800	0.137	8011.31	8000	0.141
14	801.13	800	0.141	8010.89	8000	0.136
15	801.12	800	0.14	8011.83	8000	0.148
16	801.13	800	0.141	8011.19	8000	0.14
17	801.09	800	0.136	8012.01	8000	0.15
18	801.12	800	0.14	8011.34	8000	0.142
19	801.07	800	0.134	8013.58	8000	0.169

Table 3.4: 400 LXCs Aligned on 20 Timelines (8000 LXCs)

In both the 0.1 and 1 second simulation modeling 1000 LXCs, the overhead ranges from $(0.1 - 0.23)\%$. The second pair of tests measures the same behavior with 8000 LXCs. Specifically, 8000 LXCs are modeled on 20 timelines and simulate both 0.1 and 1 virtual second. With 8000 LXCs, the overhead ranges from $(0.12 - 0.18)\%$. With respect to all the timelines, the overhead is roughly the same. This is because each timeline calls *progress(...)* 100 times in the 0.1 virtual second simulation and 1000 times in the 1 virtual second simulation. This fixed cost decreases relative to the *wallclock* time spent advancing LXCs. This amount is constant is because there is no traffic that would require more *progress(...)* calls with *smaller* advance intervals. This is explored in experiment 7.

The main takeaway from this experiment is that S3FNet-LXC is capable of supporting and managing thousands of LXCs. Like mentioned in Section 3.6, the lowest bound on simulation time is wallclock time spent advancing emulation. This experiment showcases that the overhead in emulating a fixed amount of virtual time *without traffic* is a less than half a percent of the actual wallclock time spent in emulation. This ultimately depends on the TDF and number of LXCs aligned on a single timeline.

3.8 Experiment 7: Realistic Scalability

The last experiment measures the overhead incurred on a realistic network model and compares the *progress(...)* overhead with and without heavy traffic. The model consists of 400 LXCs each aligned on one of 10 timelines, connected to each other using a ring of routers. In a 10000 μs simulation, where each LXC is dilated with TDF 300, there are both fast and slow *cross-timeline channels* as low as 50 μs. The TDF is chosen to be so high because

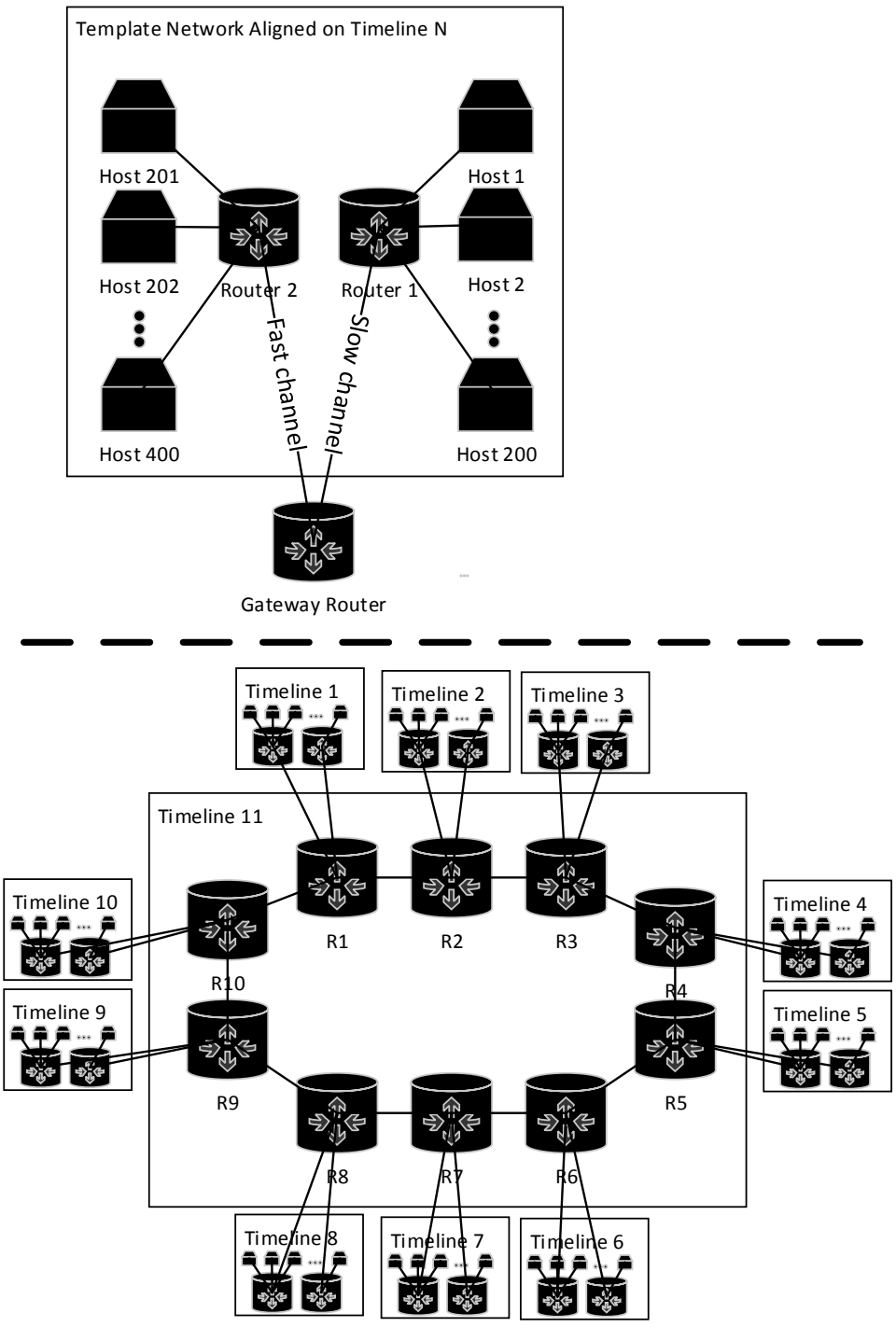

Figure 3.7: Experiment 7 Modeling 4000 LXCs

TL	Wallclock Time Spent in Progress (s)	Total Wallclock Time Spent During Emulation (s)	Progress Overhead (%)
1	1200.43	1199.52	0.076
2	1200.2	1199.52	0.057
3	1199.93	1199.52	0.034
4	1200.18	1199.52	0.055
5	1200.65	1199.52	0.094
6	1200.44	1199.52	0.077
7	1200.52	1199.52	0.083
8	1200.36	1199.52	0.07
9	1199.95	1199.52	0.036
10	1200.1	1199.52	0.048

Table 3.5: 4000 LXCs Aligned on 10 Timelines With No Traffic

TL	Wallclock Time Spent in Progress (s)	Total Wallclock Time Spent During Emulation (s)	Progress Overhead (%)
1	1353.31	1199.52	11.364
2	1456.51	1199.52	17.644
3	1385.29	1199.52	13.41
4	1318.87	1199.52	9.049
5	1371.41	1199.52	12.534
6	1340.93	1199.52	10.546
7	1377.82	1199.52	12.941
8	1404.09	1199.52	14.57
9	1313.71	1199.52	8.692
10	1314.11	1199.52	8.72

Table 3.6: 4000 LXCs Aligned on 10 Timelines With Heavy Traffic

the server incurs more advance errors than the desktop with identical TDFs. Figure 3.7 shows this topology. The simulation is ran with and without heavy traffic where all LXCs on even timelines ping all LXCs on the next odd timeline.

As Table 3.5 and 3.6 show, the simulation with traffic has a much higher overhead. This happens because the communication between the LXC Manager and Timekeeper greatly increases as LXCs advance more frequently with smaller intervals. In other words, without traffic, an LXC would normally advance from virtual time $T_1 \rightarrow T_2$ by a single interval. However, with traffic in the simulation, an LXC may advance (unfreeze and freeze) multiple times when going from $T_1 \rightarrow T_2$. This happens because a simulation will process more S3F events which are generated from emulated traffic. Consequently, *progress(...)* is called much more frequently. Note that in its current state, TimeKeeper can only wake up a timeline's worker thread (see Section 2.12) one at a time.

The main takeaway from this experiment is that there can be significant overhead during the communication between the LXC Manager and TimeKeeper. This overhead varies based on how many times and how often the LXC Manager and TimeKeeper communicate in a given interval of *virtual time*. In other words, a 1 virtual second simulation (without traffic) containing 1 LXC per timeline, where each LXC is dilated with TDF 20, means that each timeline will spend *at least* 20 wallclock seconds advancing LXCs. There is also some overhead O incurred during communication of TimeKeeper and the LXC Manager. This overhead O will vary based on how many times LXCs are advanced via *progress(...)*.

CHAPTER 4

CONCLUSION

In the last 3 chapters, this thesis introduced and described the background of simulation and emulation. The purpose of this experimental project was to utilize TimeKeeper, a tool that brings Linux processes to virtual time, to join together emulation and simulation like never before. The combination of simulation and emulation is not novel, however, previous iterations did not integrate simulation and emulation as closely S3FNet-LXC. CORE and ns-3 simulators continuously tracked observed passage of emulated time; S3FNet with OpenVZ allowed emulation and simulation to only interact at fixed points in time using barrier-based synchronization. S3FNet-LXC provides more synchronization with finer control over LXC advancement using composite synchronization. Despite all of this, there exists inherent nondeterminism in LXC advancement. The variation of the control and firing of a Linux timer is unavoidable, as is the precise amount of machine instructions that an LXC executes over repeated emulation periods. The overhead of large experiments will vary based on the amount of communication occurring between the LXC Manager and TimeKeeper. Nevertheless, the experiments in Chapter 3 illustrate that the integration of S3FNet with TimeKeeper produces the general expected behavior.

4.1 Future Work

A major limitation in the accuracy of a simulation is the accuracy of an LXC packet timestamp. That topic is explored in Section 2.13. A potential avenue to explore is bundling the *send time* of a LXC packet with the packet itself. This will remove the requirement of needing to query an LXC for a virtual timestamp. Furthermore, this will create more precise simulation results with lower TDFs as the LXC packet timestamps will be more accurate. Additionally, in order to account for the overhead ω in experiment 1, a simulation model can be extended to specify the amount of overhead incurred by an emulated application.

In its current state, not every time related aspect of the Linux kernel is represented in virtual time. By default, the traditional Linux ping program pings every 1 second. However, with the modified 3.10.9 Linux kernel, this ping happens every *wallclock* second instead of every *virtual second*. This, along with other functions such as *time*, would need to modified in the Linux kernel in order to have them operate in virtual time.

REFERENCES

[1] Core simulator. http://www.nrl.navy.mil/itd/ncs/products/core, 2015.

[2] Domain modeling language (dml). http://www.ssfnet.org/SSFdocs/dmlReference.html, 2015.

[3] Linux containers. https://linuxcontainers.org, 2015.

[4] Linux tun/tap. http://backreference.org/2010/03/26/tuntap-interface-tutorial/, 2015.

[5] Ns-3 simulator. http://www.nsnam.org/, 2015.

[6] Openvz linux containers. http://openvz.org/Main_Page, 2015.

[7] Scalable simulation framework - ssf. http://www.ssfnet.org/homePage.html/, 2015.

[8] J. Ahrenholz. Comparison of core network emulation platforms. In *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 166–171, Oct 2010.

[9] M Helsley. Lxc: Linux container tools. *IBM developerWorks Technical Library*, 2009.

[10] D Jin. S3f and s3fnet. https://s3f.iti.illinois.edu/, 2015.

[11] Dong Jin, Yuhao Zheng, Huaiyu Zhu, D.M. Nicol, and L. Winterrowd. Virtual time integration of emulation and parallel simulation. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 201–210, July 2012.

[12] Jereme Lamps, Vladimir Adam, and David M. Nicol. Conjoining emulation and network simulators on linux multiprocessors. Unpublished at time of thesis submission, To appear at SIGSIM-PADS '15, 2015.

[13] Jereme Lamps, David M. Nicol, and Matthew Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '14, pages 179–186, New York, NY, USA, 2014. ACM.

[14] D.M. Nicol, Dong Jin, and Yuhao Zheng. S3f: The scalable simulation framework revisited. In *Simulation Conference (WSC), Proceedings of the 2011 Winter*, pages 3283–3294, Dec 2011.

[15] D.M. Nicol and J. Liu. Composite synchronization in parallel discrete-event simulation. *Parallel and Distributed Systems, IEEE Transactions on*, 13(5):433–446, May 2002.

[16] Y. Zheng, D.M. Nicol, D. Jin, and N. Tanaka. A virtual time system for virtualization-based network emulations and simulations. *Journal of Simulation*, 6(3):205–213, 2012.